TU WIEN Informatics

Static and Dynamic Enforcement of Security via Relational Reasoning

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Niklas Grimm, BA
Registration Number 01652949

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Matteo Maffei

External reviewers:
Frank Piessens. KU Leuven, Belgium.
Gilles Barthe. Max Planck Institute for Security and Privacy, Germany.

Vienna, 24th January, 2021

Niklas Grimm	Matteo Maffei

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Declaration of Authorship

Niklas Grimm, BA

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 24th January, 2021

Niklas Grimm

Acknowledgements

First and foremost, I would like to thank my advisor Matteo Maffei for giving me the opportunity to write this thesis in the first place, for the very interesting research that we conducted together, and for always being motivating and encouraging throughout my studies.

I would like to express my gratitude to all the researchers with whom I collaborated in the past years. In particular, I would like Stefano Calzavara for being a great collaborator for my work in the field of web security and for being a kind host during my visits to Venice, I would like to thank Véronique Cortier for the great collaboration for my work in the area of verification of cryptographic protocols, and I would like to thank Cédric Fournet for hosting me for an exciting internship in Cambridge, and the entire F* team for the very interesting collaboration on relational reasoning in F*.

I would like to thank my colleagues, many of which I consider to be good friends, for the many interesting discussions and for making my time as PhD student enjoyable and fun. It always felt good to work in the friendly and supportive environment that they created.

I would like to thank my parents and my sister for always being helpful and for giving me all the support that I could wish for. It is great to know that I always can rely on my family. Finally, I would like to thank Barbara for being there for me and for making my life outside of work so much better.

Kurzfassung

Die statische Analyse ist ein wichtiger Baustein der Sicherheitsforschung, denn sie erlaubt es uns zu garantieren, dass formale Sicherheitseigenschaften während jeder möglichen Ausführung eines Systems eingehalten werden, bevor das System überhaupt eingesetzt wird. Viele der interessantesten Sicherheitseigenschaften lassen sich nur in einem relationalem Modell ausdrücken, d.h. durch den Vergleich mehrerer Ausführungen eines Systems. Aufgrund der Einschränkungen vorhandener Verifikationsmethoden können viele dieser Eigenschaften nur manuell überprüft werden. Die manuelle Verifizierung solcher Eigenschaften ist jedoch nicht in großem Stil durchführbar, da es sich um eine mühsame Aufgabe handelt, die ein hohes Maß an Expertenwissen erfordert. Daher stellen wir in dieser Arbeit neuartige Verifikationsverfahren vor, die eine automatisierte Analyse solcher relationaler Sicherheitseigenschaften ermöglichen. Wir basieren alle diese Verfahren auf Typsystemen, was zu effizienten und modularen Ansätzen führt. Wir präsentieren neuartige Typsysteme für die Verifikation von starken relationalen Sicherheitseigenschaften im Bereich von kryptographischen Protokollen und Web-Sicherheit und untersuchen das relationale Argumentieren in einem Beweisassistenten.

Wir entwerfen zunächst ein Verfahren für die Verifikation der Eigenschaft *Observational Equivalence* in kryptographischen Protokollen und ermöglichen damit die automatisierte Erstellung von Sicherheitsbeweisen für Protokolle, die bisher nicht durch automatisierte Verifikationsmethoden abgedeckt werden konnten.

Wir stellen dann einen Laufzeitmonitor – eine Form eines dynamischen Typsystems – für Web-Browser vor, welcher durch einfache deklarative Regeln parametrisiert wird. Wir entwerfen eine Reihe von Bedingungen für diese Regeln, die ausreichen, um starke Vertraulichkeit und Integrität von Web-Sitzungen zu garantieren.

Wir präsentieren außerdem ein Typsystem für Code von Web-Anwendungen in einem formalen Modell. Wir wenden die die Ergebnisse auf echte Web-Anwendungen an, wodurch wir neue Schwachstellen aufdecken und dann die Sicherheit von überarbeiteten Versionen der Anwendungen verifizieren.

Abschließend zeigen wir, wie der Beweisassistent F* zur Verifikation von relationalen Eigenschaften von effektvollen Programmen eingesetzt werden kann. Wir zeigen dies durch die Verifikation relationaler Eigenschaften aus verschiedenen Forschungsbereichen, einschließlich der Verifikation der Korrektheit eines Typsystems zur Informationsflusskontrolle.

Abstract

Static analysis is an important building block of security, as it allow us to guarantee that formal security properties will be preserved during any possible execution of a system, before it is even deployed. Many of the most interesting security properties can only be formulated in a relational setting, i.e., by comparing multiple executions. Due to limitations of existing tools, many of these properties can only be verified manually. However, manual verification of such properties is infeasible on a large scale, as it is a cumbersome task requiring a high degree of expert knowledge. We hence in this thesis present novel verification frameworks, that enable an automated analysis of such relational security properties. We base all of these frameworks on type systems, resulting in efficient and modular approaches. We present novel type systems for the verification of strong relational security properties in the area of cryptographic protocols and web security and study relational reasoning in a theorem prover.

We first propose a framework for the verification of observational equivalence in cryptographic protocols and establish automated proofs for protocols that previously could not be covered by automated tools.

We then propose a runtime monitor for web browsers, a form of a dynamic type system, parametrized by simple declarative policies. We design a set of constraints for these policies that is sufficient to guarantee strong web session confidentiality and integrity properties.

We also propose a type system for web application code in a formal model of the web, that enforces a strong notion of web session integrity. We apply the results to real-world web applications, uncovering novel vulnerabilities and verifying the security of fixed versions.

Finally, we show how the proof assistant F* can be used for verification of effectful relational programs. We showcase this by verifying relational properties from different research areas, including the verification of the correctness of an information control type system.

List of Publications

[CGLM17a] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A Type System for Privacy Properties. In *24th ACM Conference on Computer and Communications Security, CCS 2017*, pages 409-423. ACM, 2017.

[CGLM18a] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Equiv alence properties by typing in cryptographic branching protocols. In *Principles of Security and Trust - 7th International Conference, POST 2018*, pages 160-187. Springer, 2018.

[CFGM16a] Stefano Calzavara, Riccardo Focardi, Niklas Grimm, and Matteo Maffei. Micro-Policies for Web Session Security. In *29th IEEE Computer Security Foundations Symposium, CSF 2016*, pages 179-193. IEEE, 2016.

[CFG+20] Stefano Calzavara, Riccardo Focardi, Niklas Grimm, Matteo Maffei, and Mauro Tempesta. Language-based web session integrity. In *IEEE 33rd Computer Security Foundations Symposium, CSF 2020*, pages 107-122. IEEE, 2020.

[GMF+18] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *The 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pages 130-145. ACM 2018.

Contents

CHAPTER 1

Introduction

In today's life, computational devices and software are used for many tasks, including critical ones such as financial transactions or matters of private life, such as health. While this brings a lot of benefits in terms of usability and efficiency, problems can arise when these systems are exposed to an untrusted environment, which is for example the case with systems connected to the Internet, or when systems equipped with RFID chips (e.g., modern passports) are brought to public spaces where they can communicate with RFID readers.

For a long time the focus in development of such systems has been on improving the functionality and efficiency. However, a plethora of attacks in many different areas has shown the importance of security, and developers are more willing to allocate resources – during development as well as at runtime – in order to provide better security for the systems. During development different techniques such as code reviews or rigorous testing are used to identify potential vulnerabilities before the system is deployed. At runtime resources can be allocated to employ strong cryptographic primitives or for additional checks that can help to detect unexpected behaviour and react to it.

Although these techniques can be effective in catching vulnerabilities, due to the often infinite number of possible ways to interact with complex systems, some unexpected corner cases might still be missed and might be later exploited for attacks.

Discovering and fixing vulnerabilities in an already deployed system can be a cat-and-mouse game, and often a considerable amount of time passes before a fix is deployed on a large scale, leaving affected systems vulnerable to attacks. Even worse, many (often small) devices, do not even have the possibility to be updated, so any vulnerability will persist for the entire lifetime of the device.

1.1 Static Analysis for Security

Ideally, systems should only be deployed if the absence of vulnerabilities is guaranteed. These guarantees can be given by employing static analysis, that allows us to reason about *all* possible executions of a system, instead of only looking at single instances. This is possible by using sound overapproximations combined with mathematical principles, such as induction. Static analysis techniques have historically been used for the verification of critical systems, in order to prove important properties such as correctness or termination, and they have also been used for security properties.

Many of the techniques from the general field of verification can be – and have been – applied to security. However, the setting for security comes with its own challenges, as we typically consider the presence of an attacker who actively tries to influence the system to achieve her goals.

1.1.1 Security Properties

In order to get formal guarantees about a system, it is necessary to first formally define the desired property. The security properties covered in this thesis can be classified into two categories:

1. *Confidentiality* properties speak about what an attacker may not learn. This includes "normal" secrets such as passwords, authentication tokens, credit card numbers, but also other sensitive elements as for example a browsing history.

2. *Integrity* properties speak about what may not be influenced by an attacker. This includes that an attacker many not write to sensitive fields, as for example the recipient of a bank transaction, but also that an attacker may not influence how a vote is counted at the tally.

However, these properties come in different flavours, that allow us to express different levels of strictness. Consider the following simple example:

Listing 1.1: A simple example for confidentiality

```
1  if s0 = 0 then
2    p := s1;
3  else
4    p := p;
```

Assume we want to express that we are interested in the confidentiality of the values s_0 and s_1, and assume that the variable p is a public value, i.e., a variable that is initially known to the attacker and can be read by the attacker after execution of the program.

A straight-forward approach would be to express confidentiality as a *reachability* property. For example we could state the confidentiality of a secret s as: The variable p may never hold the value of the variable s.

One can easily verify that this property is violated for s_1 due to the assignment in line 2.

If we would use the same definition for the confidentiality of $\mathtt{s}_0$, we would conclude that it is preserved, since the value $\mathtt{s}_0$ is never written into a public variable. However, the attacker can learn something about the value $\mathtt{s}_0$, using the following observation: If after the execution of the program, the value of the public variable $\mathtt{p}$ is different from the value it had at the beginning of the program, then the value of $\mathtt{s}_0$ must be $\mathtt{0}$.

This kind of information leakage is called an *implicit flow*, opposed to the *explicit flow* that we observed for s_1.

To express confidentiality properties, that take the implicit flow into account, we use *relational* properties (also known as *hyperproperties*). These properties relate two different programs, or two runs of the same program in different environments. Intuitively, we will express the confidentiality of $\mathtt{s}_0$ by comparing two runs of the program, that only differ in the values of of $\mathtt{s}_0$ and requiring that after both runs the value of the other variables is the same, i.e., the value of $\mathtt{s}_0$ does not influence the other values.

1.1.2 Information Flow Control

The enforcement of such properties has been studied in the research area of *information flow control* (IFC). The first step in defining a policy of IFC is assigning labels to variables. For a simple confidentiality policy, secrets are labelled as high confidentiality ($\mathtt{H}$), and public values are labelled as low confidentiality ($\mathtt{L}$).

We can now express a confidentiality policy in terms of *non-interference*. Intuitively, if a value is supposed to be kept secret, it should not have any influence on the public values. We consider two runs of the same program, where in both cases the values of all variables labelled as $\mathtt{L}$ are equal, while the values of variables labelled as $\mathtt{H}$ may be arbitrarily different. We then require that after the execution the variables labelled as $\mathtt{L}$ contain the same values. Formally, such a property could be defined in the following way:

Definition 1 (Confidentiality)**.** *For two memories M and M', we write $M \approx_{\mathtt{L}} M'$, if the two memories are equal on all variables labelled as $\mathtt{L}$.*
For a program c and memories M, M', we write $M \xrightarrow{c} M'$ if evaluating c in the memory M results in the memory M'.
A program c then provides confidentiality for all variables labelled as $\mathtt{H}$, if for all memories, M_0, M_1, M_0', M_1' with $M_0 \approx_{\mathtt{L}} M_1$, $M_0 \xrightarrow{c} M_0'$ and $M_1 \xrightarrow{c} M_1'$, we have $M_0' \approx_{\mathtt{L}} M_1'$.

If we consider the example in Listing 1.1 we can then express the secrecy of $\mathtt{s}_0$ and $\mathtt{s}_1$, by assigning the label $\mathtt{H}$ to $\mathtt{s}_0$ and $\mathtt{s}_1$ and the label $\mathtt{L}$ to $\mathtt{p}$.

While this simple property in this small example is easy to verify with a quick inspection, the complexity quickly grows with the size of the program and number of variables.

1.1.3 Analysis Frameworks

At the core of every formal result is typically a formal proof. However, manually performing such a proof of a non-trivial property about a system is a difficult and cumbersome task, that requires

a high degree of expert knowledge, as it normally involves coming up with system-specific invariants.

To make obtaining formal guarantees easier, system specific analysis frameworks have been developed. Typically, the idea is to overapproximate the desired property with another property that is easier to check. For example, one can come up with a set of algorithmic checks (e.g., a type system) and show that passing these checks implies that the desired property is fulfilled. With this approach one only has to manually prove the implication. The property for a concrete instance can then be proved by performing the simpler checks (ideally automated) and relying on the implication for the actual property. Note however, that this typically comes at the price of precision since the checks are an overapproximation of the actual property.

1.2 Information Flow Control Type System

We now present a simple example for such an analysis framework. Concretely, we present a type system that is capable to enforce our information flow policy for confidentiality.

Type systems have been shown to be effective verification techniques for non-interference results. We present here a simplified version of the results shown in a seminal paper [VIS96].

Instead of showing the exact confidentiality property, we check a stronger property instead. This property overapproximates important behaviour of the program. We first assume a labelling function Γ, that assigns a label $l \in \{\mathtt{H}, \mathtt{L}\}$ to every variable.

We then aim to enforce the following two properties:

1. The content of a secret variable (labelled as H) may never be assigned to a public variable (labelled as L).
2. A public variable may not be modified if the control flow depends on a secret, i.e., if the assignment happens in a branch of a conditional that is branching over a secret value.

While we do not give a formal proof at this point, it should be intuitively clear that these properties imply our confidentiality property: By preventing assignments from secret variables to public variables, we prevent all explicit flows, and by preventing assignments to public variables in conditionals branching over a secret, we prevent implicit flows.

These checks can be represented in the following simple type system:

$$\frac{\Gamma, pc \vdash \mathtt{c}_1 \qquad \Gamma, pc \vdash \mathtt{c}_2}{\Gamma, pc \vdash \mathtt{c}_1\mathtt{;}\ \ \mathtt{c}_2}\ (\textsc{Sequence}) \qquad \frac{\Gamma(\mathtt{y}) \sqcup pc \sqsubseteq \Gamma(\mathtt{x})}{\Gamma, pc \vdash \mathtt{x}\ \mathtt{:=}\ \mathtt{y}}\ (\textsc{Assign})$$

$$\frac{pc' = pc \sqcup \Gamma(\mathtt{x}) \qquad \Gamma, pc' \vdash \mathtt{c}_t \qquad \Gamma, pc' \vdash \mathtt{c}_f}{\Gamma, pc \vdash \mathtt{if\ x\ =\ 0\ \textbf{then}\ c}_t\ \mathtt{\textbf{else}\ c}_f}\ (\textsc{If})$$

The typing judgement $\Gamma, pc \vdash \mathtt{c}$ reads as: the command $\mathtt{c}$ is well typed using the labelling function Γ and under the program counter label pc. The program counter label pc is used to track the influence of secrets on the control flow. Since at the beginning of the execution there is no dependence on any secrets, top-level typing should start with $pc = \mathtt{L}$, i.e., to prove that a program $\mathtt{c}$ fulfills the confidentiality property, we show that $\Gamma, \mathtt{L} \vdash \mathtt{c}$ can be derived using the presented rules.

The order $\sqsubseteq$ is the reflexive closure of $\mathtt{L} \sqsubseteq \mathtt{H}$ and expresses that every public value may be treated as a secret. This forms a simple lattice as shown in Fig. 1.1a. The join operator $\sqcup$ is defined as the lowest upper bound of two labels. In this case it always return the "highest" of the involved labels.

Rule (IF) treats the case of a conditional. We first compute the new program counter label $pc' = pc \sqcup \Gamma(\mathtt{x})$ which in our simple example is $\mathtt{H}$ if pc already is $\mathtt{H}$ or the label $\Gamma(\mathtt{x})$ of the guard variable is $\mathtt{H}$, and $\mathtt{L}$ otherwise. The new program counter label pc' is then used to type check both branches $\mathtt{c}_t$ and $\mathtt{c}_f$ of the conditional.

Rule (SEQUENCE) lets us type the sequence of two commands $\mathtt{c}_1\,\mathtt{;}\,\mathtt{c}_2$ by independently typing the two individual commands $\mathtt{c}_1$ and $\mathtt{c}_2$.

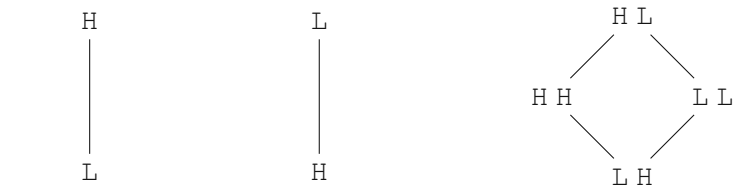
Rule (ASSIGN) is the rule that actually enforces the desired property. It states that the assignment $\mathtt{x} := \mathtt{y}$ is only well typed, if the confidentiality label of the right hand variable $\mathtt{y}$ and the program counter pc are at most as high as the one of the left hand variable $\mathtt{x}$. Concretely, this forbids assignments of of high confidentiality values to low confidentiality variables, and any assignment to low confidentiality variables under a program counter of high confidentiality.

Although this type system is just a tiny example, it suffices to show some core properties:

Automation: Type systems can typically be easily transformed into an algorithm that can be implemented in software, which helps in making the result accessible to a larger audience. Application of the algorithm to a specific program then does not require any expert knowledge, except for the initial labelling. More complicated type systems that are not syntax-driven (i.e., the syntax of the program does not uniquely identify the typing rule that should be applied in every step) require more implementation effort and potentially some heuristics and backtracking, but still the implementation can follow the typing rules very closely.

Furthermore, implementations of type systems are generally very efficient. Simple type systems – like the one presented here – have a runtime that depends linearly on the size of the program, More complex type systems sometimes require checking parts of the code multiple times under different assumptions, which can lead to higher complexity, but this is rarely an issue in practice.

Modularity: Most type systems allow for a modular approach, i.e., if we know that two parts of a program are well-typed, then also the composition is well-typed. In our example type system, this can be observed in the typing rule (SEQUENCE): If both commands can be typed in the same environment, then also the sequential composition can be typed in that environment.



(a) Lattice for confidentiality

(b) Lattice for integrity

(c) Combined lattice for confidentiality and integrity

Figure 1.1: Different common lattices for information flow control

On the one hand, this is good for efficiency, as it allows for parallelization and changes in one component don't require re-checking the entire system. On the other hand this is also helpful for designers, as single components can be developed locally and independently. The typing environment can be seen as some form of global specification that allows linking the individual components of a system.

Flexibility of Label Model: Information flow control type systems typically operate on labels, that encode the desired policy. In our example we use a simple label model, consisting only of two labels: High confidentiality (`H`) and low confidentiality (`L`). These two labels form – together with the order $\sqsubseteq$ – a lattice as pictured in Fig. 1.1a. It is possible to use other lattices in order to express different policies. For example using the lattice for integrity shown in Fig. 1.1b, where `H` stands for high and `L` for low integrity, allows us to use the same type system – without any modifications – for the verification of integrity properties. The combined lattice shown in Fig. 1.1c, where every label consists of two components – the first for confidentiality and the second for integrity – allows us to verify both confidentiality and integrity in one run of the type system. In our research, we often use more fine-grained and potentially infinitely large lattices, that can be used just like these simple lattices presented here.

1.2.1 Limitations of Information Flow Control Type Systems

Imprecision The biggest drawback in the usage of type systems for information flow control is the imprecision that is a consequence of the overapproximation used in the approach.

Consider the following example where we let $\Gamma(s) = \texttt{H}$ and $\Gamma(p) = \texttt{L}$:

Listing 1.2: Imprecision in Assignments

```
1    s := p;
2    p := s;
```

This program is a rejected because in Line 2 we assign the value of a high confidentiality variable to a low confidentiality variable, which is not allowed according to (ASSIGN). However a manual analysis of the program shows, that the secret `s` is not leaked, since the program will always assign the value of variable `p` to itself, independent of the value of `s`.

Also the treatment of the program counter can lead to imprecision, as this example shows:

Listing 1.3: Imprecision due to Program Counter

```
1  if s = 0 then
2    p0 := p1;
3  else
4    p0 := p1;
```

If we let $\Gamma(s) = \mathtt{H}$ and $\Gamma(p_0) = \Gamma(p_1) = \mathtt{L}$, then the program is rejected because in Lines 2 and 4 we have an assignment to a low confidentiality variable under a high confidentiality program counter. However, no information about the secret $\mathtt{s}$ is leaked, since in both branches of the conditional we assign the same value $\mathtt{p}_1$ to the public variable $\mathtt{p}_0$.

Attack Reconstruction While a failure of type checking can point to a problematic part of the code, reconstruction of the attack does not come automatically. Consider for example Line 4 of Listing 1.1, where typing fails due to the assignment to a low confidentiality variable in a high confidentiality context. Although in this example it is easy to see how this can be used to extract information about the secret $\mathtt{s}_0$, this attack reconstruction is not automatic and can naturally be more difficult in real-world examples.

1.2.2 Challenges

While information flow control type systems have been studied for simple while-languages [VIS96] and also for more complicated programming languages such as JavaScript [HBS16] the situation is very different for distributed systems that we are dealing with when analyzing cryptographic protocols or web settings.

The communication between multiple parties across unsecure channels brings new challenges. An attacker can read messages on the network and inject her own messages. In doing that she is not restricted by the invariants that are supposed to be enforced by labelling, e.g., she can provide low integrity data as inputs to channels that expect high integrity data. The attacker can also try to derive secret information from the communication or reflect or replay (parts of) messages by honest parties in order to provoke some undesired behaviour.

These challenges can be addressed for example by using cryptographic primitives such as encryption and signatures, or nonce checks that can ensure that a message was sent by a legitimate protocol participant that is subject to the restrictions enforced by typing. These additional ingredients require careful design of corresponding typing rules and a thorough analysis.

Although type systems for security have been used such settings [FM11], there are many security properties and fields of applications, for which previously no typing-based verification technique existed. We hence in this work push the boundaries of typing-based verification to allow for enforcement of strong relational security properties in modern applications and programs.

1.3 Contributions

Although frameworks for proofs of relational security properties exist, they are typically domain-specific and targeted at special problems. In this thesis we hence propose novel frameworks built on type systems, that allow for an efficient formal verification of relational security properties in domains where such frameworks did not previously exist, or cover new problem instances where previous approaches failed.

1.3.1 Privacy and Security Properties for Cryptographic Protocols

In Chapter 2 we present a framework for the verification of *observational equivalence* for cryptographic protocols. Observational equivalence is a relational confidentiality property on the output traces of systems that expresses that an attacker should not be able to distinguish between the execution of two different protocols even if she has full control over the public communication channel, i.e., she can read, block and inject messages.

We use this notion to express different confidentiality properties, for example the confidentiality of established keys in a key exchange protocol, confidentiality of a vote in e-voting protocols or the unlinkability of electronic passports.

As a proof technique for this property, we propose a two-layered approach: In a first step, we use a type system with an IFC core that overapproximates the execution of the protocol and collects a set of crucial observations that can be made by an attacker. In a second step we use a constraint checking procedure to show that these collected observations do not give the attacker the possibility to distinguish the two runs.

Our evaluation shows that this approach is able to efficiently verify observational equivalence for various protocols from the literature, including protocols that previously have not been successfully analyzed by automated tools.

In Chapter 3 we then extend this approach to allow typing branching protocols (i.e., protocols with non-trivial else-branches) and dynamic keys (i.e., keys generated at runtime). This lets us cover additional examples, such as the anonymous authentication protocol, while still being very efficient.

1.3.2 Browser Side Web Session Security

In Chapter 4 we consider a specific distributed system that often employs the cryptographic protocols we considered in the previous chapter: the web. Security in the web faces its own challenges, which in parts have their origin in how the web evolved. The web as we know and use it today, is the result of many ad hoc additions to a structure that was not intended to be used in the way it is today. To give one example that is crucial for security, there is no built-in support for long lived sessions, that are a building block of almost any modern web application. Hence, web applications typically rely on custom solutions built on cookies – values set by a server, stored in the browser memory and sent with each subsequent request to that same server – for this purpose. Despite their importance, guaranteeing confidentiality and integrity for these cookies is surprisingly difficult.

In this chapter we focus our security analysis on web browsers and define browser semantics as a reactive system. We then define confidentiality and integrity properties in terms of *reactive non-interference* which is a notion of non-interference tailored to reactive systems.

We propose an extension to browsers that introduces runtime checks in important interfaces of the browser. The behaviour of these runtime checks is determined by a simple declarative policy.

As a proof technique, we then design a set of overapproximating constraints for these policies and show that if a policy satisfies these constraints, then any browser equipped with runtime checks following this policy fulfills the security definitions. The shape of the policy enforced by the constraints is resembling a dynamic IFC typing approach.

This approach allows for the definition of different policies, that can be quickly checked against the set of constraints to get a guarantee for the policies' security.

By addressing web session security on the browser side, we create a mechanism that allows cautious users to protect themselves against attacks on their web sessions. These defenses can even be effective when interacting with vulnerable webpages, for which no server-side patch is deployed.

1.3.3 Server Side Web Integrity Security

In Chapter 5 we shift the focus to web servers, targeting web developers. We introduce a framework that allows for the statical verification of session integrity properties by analysis of the code of web applications. This ensures that all users of that website benefit from the strong security guarantees without the need of client side modifications.

Our goal is establishing web session integrity, a property that ensures that an attacker cannot, in any way, influence the contents of a session between an honest browser and an honest server.

We formalize small step semantics for websystems containing browsers and servers and define our integrity property as a relational property on the traces produced by such websystems. Intuitively, we identify certain assertion points in the server code, which may not be influenced by an attacker, and then require that whenever an execution of a websystem reaches such assertions in presence of an attacker, then the exact same assertions can also be reached without the presence of the attacker.

We then develop a type system for server side code and prove that successful typing implies that our session integrity property holds. The type system at its core resembles an IFC type system, but contains additional features to handle the specific web setting and session management.

Using this approach we were able to detect previously unknown vulnerabilities in web applications and verify the security of fixed versions.

1.3.4 Type Based Theorem Proving for Relational Reasoning

In Chapter 6 we showcase how the functional programming language and proof-assistant F* can be used for relational reasoning for effectful programs. As internally the strength of F* lies in

reasoning about effect-free programs, this is achieved through a process of translating effectful programs into effect-free programs.

Relying on its SMT-solving back-end, it gives us the possibility to verify relational properties with the support of automation, that can be guided by the user if necessary. Due to the expressive semantic foundation we are not impacted by the limitations of overapproximating approaches, such as type systems. As one case study, we prove the correctness of an IFC type system and show how it can be composed with the semantic approach, in order to obtain a framework that can rely on the highly efficient results of the IFC type system, while being able to fall back to a purely semantic argument for parts of the program on which the type system fails due to its imprecision.

CHAPTER 2

A Type System for Privacy Properties in Cryptographic Protocols

Abstract

Mature push button tools have emerged for checking trace properties (e.g. secrecy or authentication) of security protocols. The case of indistinguishability-based privacy properties (e.g. ballot privacy or anonymity) is more complex and constitutes an active research topic with several recent propositions of techniques and tools.
We explore a novel approach based on type systems and provide a (sound) type system for proving equivalence of protocols, for a bounded or an unbounded number of sessions. The resulting prototype implementation has been tested on various protocols of the literature. It provides a significant speed-up (by orders of magnitude) compared to tools for a bounded number of sessions and complements in terms of expressiveness other state-of-the-art tools, such as ProVerif and Tamarin: e.g., we show that our analysis technique is the first one to handle a faithful encoding of the Helios e-voting protocol in the context of an untrusted ballot box.

This chapter presents the first result of a collaboration with Véronique Cortier, Joseph Lallemand and Matteo Maffei and was published at the 24th ACM Conferecne on Computer and Communications Security (CCS'17) under the title "A Type System for Privacy Properties" [CGLM17a]. I and Joseph Lallemand contributed equally to the design of the type system and I am responsible for the reference implementation. The design of consistency procedure and the formal proofs were done mostly by Joseph Lallemand.

2.1 Introduction

Formal methods proved to be indispensable tools for the analysis of advanced cryptographic protocols such as those for key distribution [SMCB12], mobile payments [CFGT17], e-voting [DKR09,

BHM08, CEK+15], and e-health [MPR13]. In the last years, mature push-button analysis tools have emerged and have been successfully applied to many protocols from the literature in the context of *trace properties* such as authentication or confidentiality. These tools employ a variety of analysis techniques, such as model checking (e.g., Avispa [ABB+05] and Scyther [Cre08]), Horn clause resolution (e.g., ProVerif [Bla01]), term rewriting (e.g., Scyther [Cre08] and Tamarin [MSCB13]), and type systems [GJ03, BFM04, BFM05, BFM07, BBF+11, FM11, BCEM11, BCEM13, BHM14, BCEM15].

A current and very active topic is the adaptation of these techniques to the more involved case of *trace equivalence* properties. These are the natural symbolic counterpart of cryptographic indistinguishability properties, and they are at the heart of privacy properties such as ballot privacy [DKR09], untraceability [ACRR09], differential privacy [EM13], or anonymity [AF04, ACRR10]. They are also used to express stronger forms of confidentiality, such as strong secrecy [CRZ06] or game-based like properties [CC08].

Related Work. Numerous model checking-based tools have recently been proposed for the case of a bounded number of sessions, i.e., when protocols are executed a bounded number of times. These tools encompass SPEC [DT10], APTE [Che14, BDH15], Akiss [CCK12], or SAT-Equiv [CDD17]. These tools vary in the class of cryptographic primitives and the class of protocols they can consider. However, due to the complexity of the problem, they all suffer from the state explosion problem and most of them can typically analyse no more than 3-4 sessions of (relatively small) protocols, with the exception of SAT-Equiv which can more easily reach about 10 sessions. The only tools that can verify equivalence properties for an unbounded number of sessions are ProVerif [BAF08]. Maude-NPA [SEMM14], and Tamarin [BDS15]. ProVerif checks a property that is stronger than trace equivalence, namely diff equivalence, which works well in practice provided that protocols have a similar structure. However, as for trace properties, the internal design of ProVerif renders the tool unable to distinguish between exactly one session and infinitely many: this over-approximation often yields false attacks, in particular when the security of a protocol relies on the fact that some action is only performed once. Maude-NPA also checks diff-equivalence but often does not terminate. Tamarin can handle an unbounded number of sessions and is very flexible in terms of supported protocol classes but it often requires human interactions. Finally, some recent work has started to leverage type systems to enforce relational properties for programs, exploring this approach also in the context of cryptographic protocol implementations [BFG+14]: like ProVerif, the resulting tool is unable to distinguish between exactly one session and infinitely many, and furthermore it is only semi-automated, in that it often requires non-trivial lemmas to guide the tool and a specific programming discipline.

Many recent results have been obtained in the area of relational verification of programs [Ben04, Yan07, BGZ09, AGH+17, SD16, LR15, cCLRR16, GMF+18]. While these results do not target cryptographic protocols and, in particular, do not handle the semantics of cryptographic primitives or an active adversary interference with the program execution, exploring the suitability of the underlying ideas in the context of cryptographic protocols is an interesting subject of future work.

Our contribution. In this paper, we consider a novel type checking-based approach. Intuitively, a type system over-approximates protocol behavior. Due to this over-approximation, it is no

longer possible to *decide* security properties but the types typically convey sufficient information to *prove* security. Extending this approach to equivalence properties is a delicate task. Indeed, two protocols P and Q are in equivalence if (roughly) any trace of P has an equivalent trace in Q (and conversely). Over-approximating behavior may not preserve equivalence.

Instead, we develop a somewhat hybrid approach: we design a type system to over-approximate the set of possible traces and we collect the set of sent messages into *constraints*. We then propose a procedure for proving (static) equivalence of the constraints. These do not only contain sent messages but also reflect internal checks made by the protocols, which is crucial to guarantee that whenever a message is accepted by P, it is also accepted by Q (and conversely).

As a result, we provide a sound type system for proving equivalence of protocols for both a bounded and an unbounded number of sessions, or a mix of both. This is particularly convenient to analyse systems where some actions are limited (e.g., no revote, or limited access to some resource). More specifically, we show that whenever two protocols P and Q are type-checked to be equivalent, then they are in trace equivalence, for the standard notion of trace equivalence [BdNP02], against a full Dolev-Yao attacker. In particular, one advantage of our approach is that it proves security directly in a security model that is similar to the ones used by the other popular tools, in contrast to many other security proofs based on type systems. Our result holds for protocols with all standard primitives (symmetric and asymetric encryption, signatures, pairs, hash), with atomic long-term keys (no fresh keys) and no private channels. Similarly to ProVerif, we need the two protocols P and Q to have a rather similar structure.

We provide a prototype implementation of our type system, that we evaluate on several protocols of the literature. In the case of a bounded number of sessions, our tool provides a significant speed-up (less than one second to analyse a dozen of sessions while other tools typically do not answer within 12 hours, with a few exceptions). To be fair, let us emphasize that these tools can *decide* equivalence while our tool checks sufficient conditions by the means of our type system. In the case of an unbounded number of sessions, the performance of our prototype tool is comparable to ProVerif. In contrast to ProVerif, our tool can consider a mix of bounded and unbounded number of sessions. As an application, we can prove for the first time ballot privacy of the well-known Helios e-voting protocol [Adi08], without assuming a reliable channel between honest voters and the ballot box. ProVerif fails in this case as ballot privacy only holds under the assumption that honest voters vote at most once, otherwise the protocol is subject to a copy attack [Roe16]. For similar reasons, also Tamarin fails to verify this protocol.

In most of our example, only a few straightforward type annotations were needed, such as indicated which keys are supposed to be secret or public. The case of the helios protocol is more involved and requires to describe the form of encrypted ballots that can be sent by a voter.

Our prototype, the protocol models, as well as a technical report are available online [CGLM17b, CGLM17c].

2.2 Overview of our Approach

In this section, we introduce the key ideas underlying our approach on a simplified version of the Helios voting protocol. Helios [Adi08] is a verifiable voting protocol that has been used in various elections, including the election of the rector of the University of Louvain-la-Neuve. Its behavior is depicted below:

$$\begin{aligned} S \to V_i : &\quad r_i \\ V_i \to S : &\quad [\{v_i\}^{r_i, r_i'}_{\mathtt{pk}(k_s)}]_{k_i} \\ S \to V_1, \ldots, V_n : &\quad v_1, \ldots, v_n \end{aligned}$$

where $\{m\}^r_{\mathtt{pk}(k)}$ denotes the asymmetric encryption of message m with the key $\mathtt{pk}(k)$ randomized with the nonce r, and $[m]_k$ denotes the signature of m with key k. v_i is a value in the set $\{0, 1\}$, which represents the candidate V_i votes for. In the first step, the voter casts her vote, encrypted with the election's public key $\mathtt{pk}(k_s)$ and then signed. Since generating a good random number is difficult for the voter's client (typically a JavaScript run in a browser), a typical trick is to input some randomness (r_i) from the server and to add it to its own randomness (r_i'). In the second step the server outputs the tally (i.e., a randomized permutation of the valid votes received in the voting phase). Note that the original Helios protocol does not assume signed ballots. Instead, voters authenticate themselves through a login mechanism. For simplicity, we abstract this authenticated channel by a signature.

A voting protocol provides vote privacy [DKR09] if an attacker is not able to know which voter voted for which candidate. Intuitively, this can be modeled as the following trace equivalence property, which requires the attacker not to be able to distinguish A voting 0 and B voting 1 from A voting 1 and B voting 0. Notice that the attacker may control an unbounded number of voters:

$$\begin{aligned} & Voter(k_a, 0) \mid Voter(k_b, 1) \mid CompromisedVoters \mid S \\ \approx_t\ & Voter(k_a, 1) \mid Voter(k_b, 0) \mid CompromisedVoters \mid S \end{aligned}$$

Despite its simplicity, this protocol has a few interesting features that make its analysis particularly challenging. First of all, the server is supposed to discard ciphertext duplicates, otherwise a malicious eligible voter E could intercept A's ciphertext, sign it, and send it to the server [CS11], as exemplified below:

$$\begin{aligned} A \to S : &\quad [\{v_a\}^{r_a, r_a'}_{\mathtt{pk}(k_s)}]_{k_a} \\ E \to S : &\quad [\{v_a\}^{r_a, r_a'}_{\mathtt{pk}(k_s)}]_{k_e} \\ B \to S : &\quad [\{v_b\}^{r_b, r_b'}_{\mathtt{pk}(k_s)}]_{k_b} \\ S \to A, B : &\quad v_a, v_b, v_a \end{aligned}$$

This would make the two tallied results distinguishable, thereby breaking trace equivalence since $v_a, v_b, v_a \not\approx_t v_b, v_a, v_b$

Even more interestingly, each voter is supposed to be able to vote *only once*, otherwise the same attack would apply [Roe16] even if the server discards ciphertext duplicates (as the randomness used by the voter in the two ballots would be different). This makes the analysis particularly challenging, and in particular out of scope of existing cryptographic protocol analyzers like ProVerif, which abstract away from the number of protocol sessions.

With our type system, we can successfully verify the aforementioned privacy property using the following types:

$$
\begin{aligned}
& r_a : \tau_{r_a}^{\mathtt{LL},1},\ r_b : \tau_{r_b}^{\mathtt{LL},1}, r'_a : \tau_{r'_a}^{\mathtt{HH},1},\ r'_b : \tau_{r'_b}^{\mathtt{HH},1} \\
& k_a : \mathrm{key}^{\mathtt{HH}}(\{[\![\tau_0^{\mathtt{LL},1}\ ;\ \tau_1^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_a}^{\mathtt{HH},1}\}_{k_s}) \\
& k_b : \mathrm{key}^{\mathtt{HH}}(\{[\![\tau_1^{\mathtt{LL},1}\ ;\ \tau_0^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_b}^{\mathtt{HH},1}\}_{k_s}) \\
& k_s : \mathrm{key}^{\mathtt{HH}} \left(\begin{array}{l} ([\![\tau_0^{\mathtt{LL},1}\ ;\ \tau_1^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_a}^{\mathtt{HH},1}) \ \vee \\ ([\![\tau_1^{\mathtt{LL},1}\ ;\ \tau_0^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_b}^{\mathtt{HH},1}) \end{array} \right)
\end{aligned}
$$

We assume standard security labels: HH stands for high confidentiality and high integrity, HL for high confidentiality and low integrity, and LL for low confidentiality and low integrity (for simplicity, we omit the low confidentiality and high integrity type, since we do not need it in our examples). The type $\tau_i^{l,1}$ describes randomness of security label l produced by the randomness generator at position i in the program, which can be invoked at most once. $\tau_i^{l,\infty}$ is similar, with the difference that the randomness generator can be invoked an unbounded number of times. These types induce a partition on random values, in which each set contains at most one element or an unbounded number of elements, respectively. This turns out to be useful, as explained below, to type-check protocols, like Helios, in which the number of times messages of a certain shape are produced is relevant for the security of the protocol.

The type of k_a (resp. k_b) says that this key is supposed to encrypt 0 and 1 (resp. 1 and 0) on the left- and right-hand side of the equivalence relation, further describing the type of the randomness. The type of k_s inherits the two payload types, which are combined in disjunctive form. In fact, public key types implicitly convey an additional payload type, the one characterizing messages encrypted by the attacker: these are of low confidentiality and turn out to be the same on the left- and right-hand side. Key types are crucial to type-check the server code: we verify the signatures produced by A and B and can then use the ciphertext type derived from the type of k_a and k_b to infer after decryption the vote cast by A and B, respectively. While processing the other ballots, the server discards the ciphertexts produced with randomness matching the one used by A or B: given that these random values are used only once, we know that the remaining ciphertexts must come from the attacker and thus convey the same vote on the left- and on the right-hand side. This suffices to type-check the final output, since the two tallied results on the left- and right-hand side are the same, and thus fulfill trace equivalence.

The type system generates a set of constraints, which, if "consistent", suffice to prove that the protocol is trace equivalent. Intuitively, these constraints characterize the indistinguishability of the messages output by the process. The constraints generated for this simplified version of

Helios are reported below:

$$\begin{aligned} C = \{(\{&\mathtt{sign}(\mathtt{aenc}(\langle 0, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)), k_a) \sim \\ &\mathtt{sign}(\mathtt{aenc}(\langle 1, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)), k_a), \\ &\mathtt{aenc}(\langle 0, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)) \sim \mathtt{aenc}(\langle 1, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)), \\ &\mathtt{sign}(\mathtt{aenc}(\langle 1, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S)), k_b) \sim \\ &\mathtt{sign}(\mathtt{aenc}(\langle 0, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S)), k_b), \\ &\mathtt{aenc}(\langle 1, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S)) \sim \mathtt{aenc}(\langle 0, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S))\}, \\ &[x : \mathtt{LL}, y : \mathtt{LL}])\} \end{aligned}$$

These constraints are consistent if the set of left messages of the constraints is in (static) equivalence with the set of the right messages of the constraints. This is clearly the case here, since encryption hides the content of the plaintext. Just to give an example of non-consistent constraints, consider the following ones:

$$C' = \{\{\mathtt{h}(n_1) \sim \mathtt{h}(n_2),\ \ \mathtt{h}(n_1) \sim \mathtt{h}(n_1)\}\}$$

where n_1, n_2 are two confidential nonces. While the first constraint alone is consistent, since n_1 and n_2 are of high confidentiality and the attacker cannot thus distinguish between $\mathtt{h}(n_1)$ and $\mathtt{h}(n_2)$, the two constraints all together are not consistent, since the attacker can clearly notice if the two terms output by the process are the same or not. We developed a dedicated procedure to check the consistency of such constraints.

2.3 Framework

In symbolic models, security protocols are typically modeled as processes of a process algebra, such as the applied pi-calculus [AF01]. We present here a calculus close to [CCP13] inspired from the calculus underlying the ProVerif tool [Bla16].

2.3.1 Terms

Messages are modeled as terms. We assume an infinite set of names $\mathcal{N}$ for nonces, further partitioned into the set $\mathcal{FN}$ of free nonces (created by the attacker) and the set $\mathcal{BN}$ of bound nonces (created by the protocol parties), an infinite set of names $\mathcal{K}$ for keys, ranged over by k, and an infinite set of variables $\mathcal{V}$. Cryptographic primitives are modeled through a *signature* $\mathcal{F}$, that is a set of function symbols, given with their arity (that is, the number of arguments). Here, we will consider the following signature:

$$\mathcal{F}_c = \{\mathtt{pk}, \mathtt{vk}, \mathtt{enc}, \mathtt{aenc}, \mathtt{sign}, \langle\cdot,\cdot\rangle, \mathtt{h}\}$$

that models respectively public and verification key, symmetric and asymmetric encryption, concatenation and hash. The companion primitives (symmetric and asymmetric decryption, signature check, and projections) are represented by the following signature:

$$\mathcal{F}_d = \{\mathtt{dec}, \mathtt{adec}, \mathtt{checksign}, \pi_1, \pi_2\}$$

We also consider a set $\mathcal{C}$ of (public) constants (used as agents names for instance). Given a signature $\mathcal{F}$, a set of names $\mathcal{N}$ and a set of variables $\mathcal{V}$, the set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{V}, \mathcal{N})$ is the set inductively defined by applying functions to variables in $\mathcal{V}$ and names in $\mathcal{N}$. We denote by $\mathrm{names}(t)$ (resp. $\mathrm{vars}(t)$) the set of names (resp. variables) occurring in t. A term is *ground* if it does not contain variables.

Here, we will consider the set $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ of *cryptographic terms*, simply called *terms*. *Messages* are terms from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ with atomic keys, that is, a term $t \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ is a message if any subterm of t of the form $\mathtt{pk}(t')$, $\mathtt{vk}(t')$, $\mathtt{enc}(t_1, t')$, $\mathtt{aenc}(t_1, t_2)$, or $\mathtt{sign}(t_1, t')$ is such that $t' \in \mathcal{K}$ and $t_2 = \mathtt{pk}(t_2')$ with $t_2' \in \mathcal{K}$. We assume the set of variables to be split into two subsets $\mathcal{V} = \mathcal{X} \uplus \mathcal{AX}$ where $\mathcal{X}$ are variables used in processes while $\mathcal{AX}$ are variables used to store messages. An *attacker term* is a term from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{AX}, \mathcal{FN})$.

A *substitution* $\sigma = \{M_1/x_1, \ldots, M_k/x_k\}$ is a mapping from variables $x_1, \ldots, x_k \in \mathcal{V}$ to messages $M_1, \ldots, M_k$. We let $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_k\}$. We say that σ is ground if all messages $M_1, \ldots, M_k$ are ground. We let $\mathrm{names}(\sigma) = \bigcup_{1 \le i \le k} \mathrm{names}(M_i)$. The application of a substitution σ to a term t is denoted $t\sigma$ and is defined as usual.

The *evaluation* of a term t, denoted $t\downarrow$, corresponds to the application of the cryptographic primitives. For example, the decryption succeeds only if the right decryption key is used. Formally, $t\downarrow$ is recursively defined as follows.

$$
\begin{array}{rll}
u\downarrow = u & & \text{if } u \in \mathcal{N} \cup \mathcal{V} \cup \mathcal{K} \cup \mathcal{C} \\
\mathtt{pk}(t)\downarrow = \mathtt{pk}(t\downarrow) & & \text{if } t\downarrow \in \mathcal{K} \\
\mathtt{vk}(t)\downarrow = \mathtt{vk}(t\downarrow) & & \text{if } t\downarrow \in \mathcal{K} \\
\mathtt{h}(t)\downarrow = \mathtt{h}(t\downarrow) & & \text{if } t\downarrow \neq \bot \\
\langle t_1, t_2\rangle\downarrow = \langle t_1\downarrow, t_2\downarrow\rangle & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \neq \bot \\
\mathtt{enc}(t_1, t_2)\downarrow = \mathtt{enc}(t_1\downarrow, t_2\downarrow) & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \in \mathcal{K} \\
\mathtt{sign}(t_1, t_2)\downarrow = \mathtt{sign}(t_1\downarrow, t_2\downarrow) & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \in \mathcal{K} \\
\mathtt{aenc}(t_1, t_2)\downarrow = \mathtt{aenc}(t_1\downarrow, t_2\downarrow) & & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow = \mathtt{pk}(k) \\
\multicolumn{3}{c}{\text{for some } k \in \mathcal{K}}
\end{array}
$$

$$
\begin{array}{rll}
\pi_1(t)\downarrow = t_1 & & \text{if } t\downarrow = \langle t_1, t_2\rangle \\
\pi_2(t)\downarrow = t_2 & & \text{if } t\downarrow = \langle t_1, t_2\rangle \\
\mathtt{dec}(t_1, t_2)\downarrow = t_3 & & \text{if } t_1\downarrow = \mathtt{enc}(t_3, t_4) \text{ and } t_4 = t_2\downarrow \\
\mathtt{adec}(t_1, t_2)\downarrow = t_3 & & \text{if } t_1\downarrow = \mathtt{aenc}(t_3, \mathtt{pk}(t_4)) \text{ and } t_4 = t_2\downarrow \\
\mathtt{checksign}(t_1, t_2)\downarrow = t_3 & & \text{if } t_1\downarrow = \mathtt{sign}(t_3, t_4) \text{ and } t_2\downarrow = \mathtt{vk}(t_4) \\
t\downarrow = \bot & & \text{otherwise}
\end{array}
$$

Note that the evaluation of term t succeeds only if the underlying keys are atomic and always returns a message or $\bot$. We write $t =_\downarrow t'$ if $t\downarrow = t'\downarrow$.

2.3.2 Processes

Security protocols describe how messages should be exchanged between participants. We model them through a process algebra, whose syntax is displayed in Figure 2.1. We identify processes

Destructors used in processes:
$d ::= \mathtt{dec}(\cdot, k) \mid \mathtt{adec}(\cdot, k) \mid \mathtt{checksign}(\cdot, \mathtt{vk}(k)) \mid \pi_1(\cdot) \mid \pi_2(\cdot)$

Processes:

P, Q	$::=$	
	$\mathtt{0}$	
$\mid$	$\mathtt{new}\ n.P$	for $n \in \mathcal{BN}$ (n bound in P)
$\mid$	$\mathtt{out}(M).P$	
$\mid$	$\mathtt{in}(x).P$	for $x \in \mathcal{X}$ (x bound in P)
$\mid$	$P \mid Q$	
$\mid$	$\mathtt{let}\ x = d(y)\ \mathtt{in}\ P\ \mathtt{else}\ Q$	for $x, y \in \mathcal{X}$ (x bound in P)
$\mid$	$\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q$	
$\mid$	$!P$	

where M, N are messages.

Figure 2.1: Syntax for processes.

up to α-renaming, i.e., capture avoiding substitution of bound names and variables, which are defined as usual. Furthermore, we assume that all bound names and variables in the process are distinct.

A *configuration* of the system is a quadruple $(\mathcal{E}; \mathcal{P}; \phi; \sigma)$ where:

- $\mathcal{P}$ is a multiset of processes that represents the current active processes;
- $\mathcal{E}$ is a set of names, which represents the private names of the processes;
- ϕ is a substitution with $\mathrm{dom}(\phi) \subseteq \mathcal{AX}$ and for any $x \in \mathrm{dom}(\phi)$, $\phi(x)$ (also denoted $x\phi$) is a message that only contains variables in $\mathrm{dom}(\sigma)$. ϕ represents the terms already output.
- σ is a ground substitution;

The semantics of processes is given through a transition relation $\xrightarrow{\alpha}$ on the quadruples provided in Figure 2.2 (τ denotes a silent action). The relation $\xrightarrow{w}_*$ is defined as the reflexive transitive closure of $\xrightarrow{\alpha}$, where w is the concatenation of all actions. We also write equality up to silent actions $=_\tau$.

Intuitively, process $\mathtt{new}\ n.P$ creates a fresh nonce, stored in $\mathcal{E}$, and behaves like P. Process $\mathtt{out}(M).P$ emits M and behaves like P. Process $\mathtt{in}(x).P$ inputs any term computed by the attacker provided it evaluates as a message and then behaves like P. Process $P \mid Q$ corresponds to the parallel composition of P and Q. Process $\mathtt{let}\ x = d(y)\ \mathtt{in}\ P\ \mathtt{else}\ Q$ behaves like P in which x is replaced by $d(y)$ if $d(y)$ can be successfully evaluated and behaves like Q otherwise. Process $\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q$ behaves like P if M and N correspond to two equal messages and behaves like Q otherwise. The replicated process $!P$ behaves as an unbounded number of copies of P.

$$(\mathcal{E}, \{P_1 \mid P_2\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}\{P_1, P_2\} \cup \mathcal{P}; \phi; \sigma) \quad \text{PAR}$$

$$(\mathcal{E}, \{\mathtt{0}\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}, \mathcal{P}; \phi; \sigma) \quad \text{ZERO}$$

$$(\mathcal{E}, \{\mathtt{new}\ n.P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}, \{P\} \cup \mathcal{P}; \phi; \sigma) \quad \text{NEW}$$

$$(\mathcal{E}, \{\mathtt{out}(t).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\mathtt{new}\ ax_n.\mathtt{out}(ax_n)} (\mathcal{E}, \{P\} \cup \mathcal{P}; \phi \cup \{t/ax_n\}; \sigma) \quad \text{OUT}$$
if $t\sigma$ is a ground term, $ax_n \in \mathcal{AX}$ and $n = |\phi| + 1$

$$(\mathcal{E}, \{\mathtt{in}(x).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\mathtt{in}(R)} (\mathcal{E}, \{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(R\phi\sigma)\downarrow /x\}) \quad \text{IN}$$
if R is an attacker term such that $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$, and $(R\phi\sigma)\downarrow \neq \bot$

$$(\mathcal{E}, \{\mathtt{let}\ x = d(M)\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}, \{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(d(M)\sigma)\downarrow /x\}) \quad \text{LET-IN}$$
if $M\sigma$ is ground and $d(M\sigma)\downarrow \neq \bot$

$$(\mathcal{E}, \{\mathtt{let}\ x = d(M)\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}, \{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \text{LET-ELSE}$$
if $M\sigma$ is ground and $d(M\sigma)\downarrow = \bot$

$$(\mathcal{E}, \{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}, \{P\} \cup \mathcal{P}; \phi; \sigma) \quad \text{IF-THEN}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground, $M\sigma = N\sigma$

$$(\mathcal{E}, \{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}, \{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \text{IF-ELSE}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground and $M\sigma \neq N\sigma$

$$(\mathcal{E}, \{!P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{E}, \{P, !P\} \cup \mathcal{P}; \phi; \sigma) \quad \text{REPL}$$

Figure 2.2: Semantics

A *trace* of a process P is any possible sequence of transitions in the presence of an attacker that may read, forge, and send messages. Formally, the set of traces $\mathrm{trace}(P)$ is defined as follows.

$$\mathrm{trace}(P) = \{(w, \mathtt{new}\ \mathcal{E}.\phi, \sigma) | (\emptyset; \{P\}; \emptyset; \emptyset) \xrightarrow{w}_* (\mathcal{E}; \mathcal{P}; \phi; \sigma)\}$$

Example 1. *Consider the Helios protocol presented in Section 2.2. For simplicity, we describe here a simplified version with only two (honest) voters A and B and a voting server S. This (simplified) protocol can be modeled by the process:*

$$\mathtt{new}\ r_a.Voter(k_a, v_a, r_a) \mid \mathtt{new}\ r_b.Voter(k_b, v_b, r_b) \mid P_S$$

where $Voter(k, v, r)$ represents voter k willing to vote for v using randomness r while P_S represents the voting server. $Voter(k, v, r)$ simply outputs a signed encrypted vote.

$$Voter(k, v, r) = \mathtt{out}(\mathtt{sign}(\mathtt{aenc}(\langle v, r\rangle, \mathtt{pk}(k_S)), k))$$

The voting server receives ballots from A and B and then outputs the decrypted ballots, after

some mixing.

$$\begin{aligned} P_S = & \mathtt{in}(x_1).\mathtt{in}(x_2). \\ & \mathtt{let}\ y_1 = \mathtt{checksign}(x_1, \mathtt{vk}(k_a))\ \mathtt{in} \\ & \mathtt{let}\ y_2 = \mathtt{checksign}(x_2, \mathtt{vk}(k_b))\ \mathtt{in} \\ & \mathtt{let}\ z_1 = \mathtt{adec}(y_1, k_s)\ \mathtt{in} \quad \mathtt{let}\ z_1' = \pi_1(z_1)\ \mathtt{in} \\ & \mathtt{let}\ z_2 = \mathtt{adec}(y_2, k_s)\ \mathtt{in} \quad \mathtt{let}\ z_2' = \pi_1(z_2)\ \mathtt{in} \\ & \quad (\mathtt{out}(z_1') \mid \mathtt{out}(z_2')) \end{aligned}$$

2.3.3 Equivalence

When processes evolve, sent messages are stored in a substitution ϕ while private names are stored in $\mathcal{E}$. A *frame* is simply an expression of the form $\mathtt{new}\ \mathcal{E}.\phi$ where $\mathrm{dom}(\phi) \subseteq \mathcal{AX}$. We define $\mathrm{dom}(\mathtt{new}\ \mathcal{E}.\phi)$ as $\mathrm{dom}(\phi)$. Intuitively, a frame represents the knowledge of an attacker.

Intuitively, two sequences of messages are indistinguishable to an attacker if he cannot perform any test that could distinguish them. This is typically modeled as static equivalence [AF01]. Here, we consider of variant of [AF01] where the attacker is also given the ability to observe when the evaluation of a term fails, as defined for example in [CCP13].

Definition 1 (Static Equivalence)**.** *Two ground frames* $\mathtt{new}\ \mathcal{E}.\phi$ *and* $\mathtt{new}\ \mathcal{E}'.\phi'$ *are statically equivalent if and only if they have the same domain, and for all attacker terms* R, S *with variables in* $\mathrm{dom}(\phi) = \mathrm{dom}(\phi')$*, we have*

$$(R\phi =_{\downarrow} S\phi) \iff (R\phi' =_{\downarrow} S\phi')$$

Then two processes P and Q are in equivalence if no matter how the adversary interacts with P, a similar interaction may happen with Q, with equivalent resulting frames.

Definition 2 (Trace Equivalence)**.** *Let* P*,* Q *be two processes. We write* $P \sqsubseteq_t Q$ *if for all* $(s, \psi, \sigma) \in \mathrm{trace}(P)$*, there exists* $(s', \psi', \sigma') \in \mathrm{trace}(Q)$ *such that* $s =_\tau s'$ *and* $\psi\sigma$ *and* $\psi'\sigma'$ *are statically equivalent. We say that* P *and* Q *are trace equivalent, and we write* $P \approx_t Q$*, if* $P \sqsubseteq_t Q$ *and* $Q \sqsubseteq_t P$*.*

Note that this definition already includes the attacker's behavior, since processes may input any message forged by the attacker.

Example 2. *As explained in Section 2.2, ballot privacy is typically modeled as an equivalence property [DKR09] that requires that an attacker cannot distinguish when Alice is voting* 0 *and Bob is voting* 1 *from the scenario where the two votes are swapped.*

Continuing Example 1, ballot privacy of Helios can be expressed as follows:

$$\begin{aligned} & \mathtt{new}\ r_a.Voter(k_a, 0, r_a) \mid \mathtt{new}\ r_b.Voter(k_b, 1, r_b) \mid P_S \\ \approx_t\ & \mathtt{new}\ r_a.Voter(k_a, 1, r_a) \mid \mathtt{new}\ r_b.Voter(k_b, 0, r_b) \mid P_S \end{aligned}$$

$$
\begin{array}{lll}
l & ::= & \mathtt{LL} \mid \mathtt{HL} \mid \mathtt{HH} \\
T & ::= & l \mid T * T \mid \mathrm{key}^{l}(T) \mid (T)_k \mid \{T\}_k \\
& \mid & [\![\tau_n^{l,a} \,;\, \tau_m^{l',a}]\!] \text{ with } a \in \{1, \infty\} \mid T \vee T
\end{array}
$$

Figure 2.3: Types for terms (selected)

2.4 Typing

We now introduce a type system to statically check trace equivalence between processes. Our typing judgements thus capture properties of pairs of terms or processes, which we will refer to as *left* and *right* term or process, respectively.

2.4.1 Types

A selection of the types for messages are defined in Figure 2.3 and explained below. We assume three security labels (namely, $\mathtt{HH}, \mathtt{HL}, \mathtt{LL}$), ranged over by l, whose first (resp. second) component denotes the confidentiality (resp. integrity) level. Intuitively, messages of high confidentiality cannot be learned by the attacker, while messages of high integrity cannot originate from the attacker. Pair types describe the type of their components, as usual. Type $\mathrm{key}^{l}(T)$ describes keys of security level l used to encrypt (or sign) messages of type T. The type $(T)_k$ (resp. $\{T\}_k$) describes symmetric (resp. asymmetric) encryptions with key k of a message of type T. The type $\tau_i^{l,a}$ describes nonces and constants of security level l: the label a ranges over $\{\infty, 1\}$, denoting whether the nonce is bound within a replication or not (constants are always typed with $a = 1$). We assume a different identifier i for each constant and restriction in the process. The type $\tau_i^{l,1}$ is populated by a single name, (i.e., i describes a constant or a non-replicated nonce) and $\tau_i^{l,\infty}$ is a special type, that is instantiated to $\tau_{i_j}^{l,1}$ in the jth replication of the process. Type $[\![\tau_n^{l,a} \,;\, \tau_m^{l',a}]\!]$ is a refinement type that restricts the set of values which can be taken by a message to values of type $\tau_n^{l,a}$ on the left and type $\tau_m^{l',a}$ on the right. For a refinement type $[\![\tau_n^{l,a} \,;\, \tau_n^{l,a}]\!]$ with equal types on both sides we simply write $\tau_n^{l,a}$. Messages of type $T \vee T'$ are messages that can have type T or type T'.

2.4.2 Constraints

When typing messages, we generate constraints of the form $(M \sim N)$, meaning that the attacker sees M and N in the left and right process, respectively, and these two messages are thus required to be indistinguishable.

2.4.3 Typing Messages

Typing judgments are parametrized over a typing environment Γ, which is a list of mappings from names and variables to types. The typing judgement for messages is of the form the form $\Gamma \vdash M \sim N : T \to c$ which reads as follows: under the environment Γ, M and N are of type T and either this is a high confidentiality type (i.e., M and N are not disclosed to the attacker) or M and N are indistinguishable for the attacker assuming the set of constraints c holds true. We

$$\text{(TNONCE)}\ \frac{\Gamma(n) = \tau_n^{l,a} \qquad \Gamma(m) = \tau_m^{l,a} \qquad l \in \{\mathtt{HH}, \mathtt{HL}\}}{\Gamma \vdash n \sim m : l \to \emptyset} \qquad \text{(TNONCEL)}\ \frac{\Gamma(n) = \tau_n^{\mathtt{LL},a}}{\Gamma \vdash n \sim n : \mathtt{LL} \to \emptyset}$$

$$\text{(TCSTFN)}\ \frac{a \in \mathcal{C} \cup \mathcal{FN}}{\Gamma \vdash a \sim a : \mathtt{LL} \to \emptyset} \qquad \text{(TPUBKEY)}\ \frac{k \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{pk}(k) \sim \mathtt{pk}(k) : \mathtt{LL} \to \emptyset} \qquad \text{(TVKEY)}\ \frac{k \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{vk}(k) \sim \mathtt{vk}(k) : \mathtt{LL} \to \emptyset}$$

$$\text{(TKEY)}\ \frac{\Gamma(k) = T}{\Gamma \vdash k \sim k : T \to \emptyset} \qquad \text{(TVAR)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x \sim x : T \to \emptyset}$$

$$\text{(TPAIR)}\ \frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c'}{\Gamma \vdash \langle M, M' \rangle \sim \langle N, N' \rangle : T * T' \to c \cup c'}$$

$$\text{(TENC)}\ \frac{\Gamma \vdash M \sim N : T \to c}{\Gamma \vdash \mathtt{enc}(M, k) \sim \mathtt{enc}(N, k) : (T)_k \to c} \qquad \text{(TENCH)}\ \frac{\Gamma \vdash M \sim N : (T)_k \to c \qquad \Gamma(k) = \mathrm{key}^{\mathtt{HH}}(T)}{\Gamma \vdash M \sim N : \mathtt{LL} \to c \cup \{M \sim N\}}$$

$$\text{(TENCL)}\ \frac{\Gamma \vdash M \sim N : (\mathtt{LL})_k \to c \qquad \Gamma(k) = \mathrm{key}^{\mathtt{LL}}(T)}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}$$

$$\text{(TAENC)}\ \frac{\Gamma \vdash M \sim N : T \to c}{\Gamma \vdash \mathtt{aenc}(M, \mathtt{pk}(k)) \sim \mathtt{aenc}(N, \mathtt{pk}(k)) : \{T\}_k \to c}$$

$$\text{(TAENCH)}\ \frac{\Gamma \vdash M \sim N : \{T\}_k \to c \qquad \Gamma(k) = \mathrm{key}^{\mathtt{HH}}(T)}{\Gamma \vdash M \sim N : \mathtt{LL} \to c \cup \{M \sim N\}}$$

$$\text{(TAENCL)}\ \frac{\Gamma \vdash M \sim N : \{\mathtt{LL}\}_k \to c \qquad k \in \mathrm{dom}(\Gamma)}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}$$

$$\text{(TSIGNH)}\ \frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c' \qquad \Gamma(k) = \mathrm{key}^{\mathtt{HH}}(T)}{\Gamma \vdash \mathtt{sign}(M, k) \sim \mathtt{sign}(N, k) : \mathtt{LL} \to c \cup c' \cup \{\mathtt{sign}(M, k) \sim \mathtt{sign}(N, k)\}}$$

Figure 2.4: Rules for Messages (1)

$$\text{(TSignL)}\quad \frac{\Gamma \vdash M \sim N : \mathtt{LL} \to c \qquad \Gamma(k) = \mathrm{key}^{\mathtt{LL}}(T)}{\Gamma \vdash \mathtt{sign}(M,k) \sim \mathtt{sign}(N,k) : \mathtt{LL} \to c}$$

$$\text{(THash)}\quad \frac{\mathrm{names}(M) \cup \mathrm{names}(N) \cup \mathrm{vars}(M) \cup \mathrm{vars}(N) \subseteq \mathrm{dom}(\Gamma) \cup \mathcal{FN}}{\Gamma \vdash \mathtt{h}(M) \sim \mathtt{h}(N) : \mathtt{LL} \to \{\mathtt{h}(M) \sim \mathtt{h}(N)\}}$$

$$\text{(THashL)}\quad \frac{\Gamma \vdash M \sim N : \mathtt{LL} \to c}{\Gamma \vdash \mathtt{h}(M) \sim \mathtt{h}(N) : \mathtt{LL} \to c}$$

$$\text{(THigh)}\quad \frac{\mathrm{names}(M) \cup \mathrm{names}(N) \cup \mathrm{vars}(M) \cup \mathrm{vars}(N) \subseteq \mathrm{dom}(\Gamma) \cup \mathcal{FN}}{\Gamma \vdash M \sim N : \mathtt{HL} \to \emptyset}$$

$$\text{(TSub)}\quad \frac{\Gamma \vdash M \sim N : T' \to c \qquad T' <: T}{\Gamma \vdash M \sim N : T \to c} \qquad \text{(TOr)}\quad \frac{\Gamma \vdash M \sim N : T \to c}{\Gamma \vdash M \sim N : T \vee T' \to c}$$

$$\text{(TLR}^1\text{)}\quad \frac{\Gamma(m) = \tau_m^{l,1} \quad \text{or} \quad m \in \mathcal{FN} \cup \mathcal{C} \wedge l = \mathtt{LL} \qquad \Gamma(n) = \tau_n^{l',1} \quad \text{or} \quad n \in \mathcal{FN} \cup \mathcal{C} \wedge l' = \mathtt{LL}}{\Gamma \vdash m \sim n : [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!] \to \emptyset}$$

$$\text{(TLR}^\infty\text{)}\quad \frac{\Gamma(m) = \tau_m^{l,\infty} \qquad \Gamma(n) = \tau_n^{l',\infty}}{\Gamma \vdash m \sim n : [\![\tau_m^{l,\infty}\,;\,\tau_n^{l',\infty}]\!] \to \emptyset}$$

$$\text{(TLR')}\quad \frac{\Gamma \vdash M \sim N : [\![\tau_m^{l,a}\,;\,\tau_n^{l,a}]\!] \to c \qquad l \in \{\mathtt{HL}, \mathtt{HH}\}}{\Gamma \vdash M \sim N : l \to c} \qquad \text{(TLRL')}\quad \frac{\Gamma \vdash M \sim N : [\![\tau_n^{\mathtt{LL},a}\,;\,\tau_n^{\mathtt{LL},a}]\!] \to c}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}$$

$$\text{(TLRVar)}\quad \frac{\Gamma \vdash x \sim x : [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!] \to \emptyset \qquad \Gamma \vdash y \sim y : [\![\tau_{m'}^{l'',1}\,;\,\tau_{n'}^{l''',1}]\!] \to \emptyset}{\Gamma \vdash x \sim y : [\![\tau_m^{l,1}\,;\,\tau_{n'}^{l''',1}]\!] \to \emptyset}$$

Figure 2.5: Rules for Messages (2)

present an excerpt of the typing rules for messages in Figures 2.4 and 2.5 and comment on them in the following.

Confidential nonces (i.e. nonces with label $l = \texttt{HH}$ or $l = \texttt{HL}$) are typed with their label from the typing environment. As the attacker may not observer them, they may be different in the left and the right message and we do not add any constraints (TNONCE). Public terms are given type LL if they are the same in the left and the right message (TNONCEL, TCSTFN, TPUBKEY, TVKEY). We require keys and variables to be the same in the two processes, deriving their type from the environment (TKEY and TVAR). The rule for pairs operates recursively component-wise (TPAIR).

For symmetric key encryptions (TENC), we have to make sure that the payload type matches the key type (which is achieved by rule TENCH). We add the generated ciphertext to the set of constraints, because even though the attacker cannot read the plaintext, he can perform an equality check on the ciphertext that he observed. If we type an encryption with a key that is of low confidentiality (i.e., the attacker has access to it), then we need to make sure the payload is of type LL, because the attacker can simply decrypt the message and recover the plaintext (TENCL). The rules for asymmetric encryption are the same, with the only difference that we can always chose to ignore the key type and use type LL to check the payload. This allows us to type messages produced by the attacker, which has access to the public key but does not need to respect its type. Signatures are also handled similarly, the difference here is that we need to type the payload with LL even if an honest key is used, as the signature does not hide the content. The first typing rule for hashes (THASH) gives them type LL and adds the term to the constraints, without looking at the arguments of the hash function: intuitively this is justified, because the hash function makes it impossible to recover the argument. The second rule (THASHL) gives type LL only if we can also give type LL to the argument of the hash function, but does not add any constraints on its own, it is just passing on the constraints created for the arguments. This means we are typing the message as if the hash function would not have been applied and use the message without the hash, which is a strictly stronger result. Both rules have their applications: while the former has to be used whenever we hash a secret, the latter may be useful to avoid the creation of unnecessary constraints when hashing terms like constants or public nonces. Rule THIGH states that we can give type HL to every message, which intuitively means that we can treat every message as if it were confidential. Rule TSUB allows us to type messages according to the subtyping relation, which is standard and defined in Figure 2.6. Rule TOR allows us to give a union type to messages, if they are typable with at least one of the two types. TLR^1 and TLR^∞ are the introduction rules for refinement types, while TLR' and TLRL' are the corresponding elimination rules. Finally, TLRVAR allows to derive a new refinement type for two variables for which we have singleton refinement types, by taking the left refinement of the left variable and the right refinement of the right variable. We will see application of this rule in the e-voting protocol, where we use it to combine A's vote (0 on the left, 1 on the right) and B's vote (1 on the left, 0 on the right), into a message that is the same on both sides.

$$\frac{}{T <: T}\ (\textsc{SRefl}) \qquad \frac{}{T <: \mathtt{HL}}\ (\textsc{SHigh}) \qquad \frac{T <: T' \qquad T' <: T''}{T <: T''}\ (\textsc{STrans}) \qquad \frac{}{\mathtt{LL} * \mathtt{LL} <: \mathtt{LL}}\ (\textsc{SPairL}) \qquad \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}\ (\textsc{SPair})$$

$$\frac{}{\mathtt{HH} * T <: \mathtt{HH}}\ (\textsc{SPairS}) \qquad \frac{}{T * \mathtt{HH} <: \mathtt{HH}}\ (\textsc{SPairS'}) \qquad \frac{}{\mathrm{key}^l(T) <: l}\ (\textsc{SKey}) \qquad \frac{T <: T'}{(T)_k <: (T')_k}\ (\textsc{SEnc}) \qquad \frac{T <: T'}{\{T\}_k <: \{T'\}_k}\ (\textsc{SAEnc})$$

Figure 2.6: Subtyping Rules

2.4.4 Typing Processes

The typing judgement for processes is of the form $\Gamma \vdash P \sim Q \to C$ and can be interpreted as follows: If two processes P and Q can be typed in Γ and if the generated constraint set C is consistent, then P and Q are trace equivalent. We assume in this section that P and Q do not contain replication and that variables and names are renamed to avoid any capture. We also assume processes to be given with type annotations for nonces.

When typing processes, the typing environment Γ is passed down and extended from the root towards the leafs of the syntax tree of the process, i.e., following the execution semantics. The generated constraints C however, are passed up from the leafs towards the root, so that at the root we get all generated constraints, modeling the attacker's global view on the process execution.

More precisely, each possible execution path of the process - there may be multiple paths because of conditionals - creates its own set of constraints c together with the typing environment Γ that contains types for all names and variables appearing in c. Hence a *constraint set* C is a set elements of the form (c, Γ) for a set of constraints c. The typing environments are required in the constraint checking procedure, as they helps us to be more precise when checking the consistency of constraints.

An excerpt of our typing rules for processes is presented in Figure 2.7 and explained in the following. Rule PZero copies the current typing environment in the constraints and checks the well-formedness of the environment ($\Gamma \vdash \diamond$), which is defined as expected. Messages output on the network are possibly learned by the attacker, so they have to be of type LL (POut). The generated constraints are added to each element of the constraint set for the continuation process, using the operator $\cup_\forall$ defined as

$$C \cup_\forall c' := \{(c \cup c', \Gamma) \mid (c, \Gamma) \in C\}\,.$$

Conversely, messages input from the network are given type LL (PIn). Rule PNew introduces a new nonce, which may be used in the continuation processes. While typing parallel composition (PPar), we type the individual subprocesses and take the product union of the generated constraint

(PZERO)
$$\frac{\Gamma \vdash \diamond \qquad \Gamma \text{ does not contain union types}}{\Gamma \vdash \mathtt{0} \sim \mathtt{0} \to (\emptyset, \Gamma)}$$

(POUT)
$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c}{\Gamma \vdash \mathtt{out}(M).P \sim \mathtt{out}(N).Q \to C \cup_\forall c}$$

(PIN)
$$\frac{\Gamma, x : \mathtt{LL} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{in}(x).P \sim \mathtt{in}(x).Q \to C}$$

(PNEW)
$$\frac{\Gamma, n : \tau_n^{l,a} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{new}\ n : \tau_n^{l,a}.P \sim \mathtt{new}\ n : \tau_n^{l,a}.Q \to C}$$

(PPAR)
$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash P \mid P' \sim Q \mid Q' \to C \cup_\times C'}$$

(POR)
$$\frac{\Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma, x : T' \vdash P \sim Q \to C'}{\Gamma, x : T \vee T' \vdash P \sim Q \to C \cup C'}$$

(PLET)
$$\frac{\Gamma \vdash d(y) : T \qquad \Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash \mathtt{let}\ x = d(y)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = d(y)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to C \cup C'}$$

(PLETLR)
$$\frac{\Gamma(y) = [\![\tau_n^{l,a}\,;\,\tau_m^{l',a}]\!] \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash \mathtt{let}\ x = d(y)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = d(y)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to C'}$$

(PIFL)
$$\frac{\begin{array}{c}\Gamma \vdash P \sim Q \to C \\ \Gamma \vdash P' \sim Q' \to C' \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c \qquad \Gamma \vdash M' \sim N' : \mathtt{LL} \to c'\end{array}}{\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to (C \cup C') \cup_\forall (c \cup c')}$$

(PIFLR)
$$\frac{\begin{array}{c}\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!] \to \emptyset \qquad \Gamma \vdash M_2 \sim N_2 : [\![\tau_{m'}^{l'',1}\,;\,\tau_{n'}^{l''',1}]\!] \to \emptyset \\ b = (\tau_m^{l,1} \stackrel{?}{=} \tau_{m'}^{l'',1}) \qquad b' = (\tau_n^{l',1} \stackrel{?}{=} \tau_{n'}^{l''',1}) \qquad \Gamma \vdash P_b \sim Q_{b'} \to C\end{array}}{\Gamma \vdash \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P_\top\ \mathtt{else}\ P_\bot \sim \mathtt{if}\ N_1 = N_2\ \mathtt{then}\ Q_\top\ \mathtt{else}\ Q_\bot \to C}$$

(PIFS)
$$\frac{\Gamma \vdash P' \sim Q' \to C' \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c \qquad \Gamma \vdash M' \sim N' : \mathtt{HH} \to c'}{\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C'}$$

(PIFLR*)
$$\frac{\begin{array}{c}\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,\infty}\,;\,\tau_n^{l',\infty}]\!] \to \emptyset \qquad \Gamma \vdash M_2 \sim N_2 : [\![\tau_m^{l,\infty}\,;\,\tau_n^{l',\infty}]\!] \to \emptyset \\ \Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'\end{array}}{\Gamma \vdash \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N_1 = N_2\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C \cup C'}$$

Figure 2.7: Rules for processes

sets as the new constraint set. The *product union* of constraint sets is defined as

$$C \cup_\times C' := \{(c \cup c', \Gamma \cup \Gamma') \mid (c, \Gamma) \in C \wedge (c', \Gamma') \in C' \wedge \Gamma, \Gamma' \text{ are compatible}\}$$

where *compatible* environments are those that agree on the type of all arguments of the shared domain. This operation models the fact that a process $P \mid P'$ can have every trace that is a combination of any trace of P with any trace of P'. The branches that are discarded due to incompatible environments correspond to impossible executions (e.g., taking the left branch in P and the right branch in P' in two conditionals with the same guard). POR is the elimination rule for union types, which requires the continuation process to be well-typed with both types.

To ensure that the destructor application fails or succeeds equally in the two processes, we allow only the same destructor to be applied to the same variable in both processes (PLET). As usual, we then type-check the then as well as the else branch and then take the union of the corresponding constraints. The typing rules for destructors are presented in Figure 2.8. These are mostly standard: for instance, after decryption, the type of the payload is determined by the one of the decryption key, as long as this is of high integrity (DDECH). We can as well exploit strong types for ciphertexts, typically introduced by verifying a surrounding signature (see, e.g., the types for Helios) to derive the type of the payload (DDECT). In the case of public key encryption, we have to be careful, since the public encryption key is accessible to the attacker: we thus give the payload type $T \vee \mathtt{LL}$ (rule DADECH). For operations involving corrupted keys (label LL) we know that the payload is public and hence give the derived message type LL.

In the special case in which we know that the concrete value of the argument of the destructor application is a nonce or constant due to a refinement type, and we know statically that any destructor application will fail, we only need to type-check the else branch (PLETLR). As for destructor applications, the difficulty while typing conditionals is to make sure that the same branch is taken in both processes (PIFL). To ensure this we use a trick: We type both the left and the right operands of the conditional with type LL and add both generated sets of constraints to the constraint set. Intuitively, this means that the attacker could perform the equality test himself, since the guard is of type LL, which means that the conditional must take the same branch on the left and on the right. In the special case in which we can statically determine the concrete value of the terms in the conditional (because the corresponding type is populated by a singleton), we have to typecheck only the single combination of branches that will be executed (PIFLR). Another special case is if the messages on the right are of type HH and the ones on the left of type LL. As a secret of high integrity can never be equal to a public value of low integrity, we know that both processes will take the else branch (PIFS). This rule is crucial, since it may allow us to prune the low typing branch of asymmetric decryption. The last special case for conditionals is when we have a refinement type with replication for both operands of the equality check (PIFLR*). Although we know that the nonces on both sides are of the same type and hence both are elements of the same set, we cannot assume that they are equal, as the sets are infinite, unlike in rule PIFLR. Yet, concrete instantiations of nonces will have the same index for the left and the right process. This is because we check for a variant of diff-equivalence. This ensures that the equality check always yields the same result in the two processes. All these special cases highlight how

$$\text{(DDecH)}\ \frac{\Gamma(k) = \text{key}^{\text{HH}}(T) \qquad \Gamma(x) = \text{LL}}{\Gamma \vdash \texttt{dec}(x,k) : T} \qquad \text{(DDecL)}\ \frac{\Gamma(k) = \text{key}^{\text{LL}}(T) \qquad \Gamma(x) = \text{LL}}{\Gamma \vdash \texttt{dec}(x,k) : \text{LL}}$$

$$\text{(DDecT)}\ \frac{\Gamma(x) = (T)_k}{\Gamma \vdash \texttt{dec}(x,k) : T} \qquad \text{(DAdecH)}\ \frac{\Gamma(k) = \text{key}^{\text{HH}}(T) \qquad \Gamma(x) = \text{LL}}{\Gamma \vdash \texttt{adec}(x,k) : T \vee \text{LL}}$$

$$\text{(DAdecL)}\ \frac{\Gamma(k) = \text{key}^{\text{LL}}(T) \qquad \Gamma(x) = \text{LL}}{\Gamma \vdash \texttt{adec}(x,k) : \text{LL}} \qquad \text{(DAdecT)}\ \frac{\Gamma(x) = \{T\}_k}{\Gamma \vdash \texttt{adec}(x,k) : T}$$

$$\text{(DCheckH)}\ \frac{\Gamma(k) = \text{key}^{\text{HH}}(T) \qquad \Gamma(x) = \text{LL}}{\Gamma \vdash \texttt{checksign}(x, \texttt{vk}(k)) : T} \qquad \text{(DCheckL)}\ \frac{\Gamma(k) = \text{key}^{\text{LL}}(T) \qquad \Gamma(x) = \text{LL}}{\Gamma \vdash \texttt{checksign}(x, \texttt{vk}(k)) : \text{LL}} \qquad \text{(DFst)}\ \frac{\Gamma(x) = T * T'}{\Gamma \vdash \pi_1(x) : T}$$

$$\text{(DSnd)}\ \frac{\Gamma(x) = T * T'}{\Gamma \vdash \pi_2(x) : T'} \qquad \text{(DFstL)}\ \frac{\Gamma(x) = \text{LL}}{\Gamma \vdash \pi_1(x) : \text{LL}} \qquad \text{(DSndL)}\ \frac{\Gamma(x) = \text{LL}}{\Gamma \vdash \pi_2(x) : \text{LL}}$$

Figure 2.8: Destructor Rules

a careful treatment of names in terms of equivalence classes (statically captured by types) is a powerful device to enhance the expressiveness of the analysis.

Finally, notice that we do not have any typing rule for replication: this is in line with our general idea of typing a bounded number of sessions and then extending this result to the unbounded case in the constraint checking phase, as detailed in Section 2.6.

2.5 Consistency of Constraints

Our type system guarantees trace equivalence of two processes only if the generated constraints are *consistent*. In this section we give a slightly simplified definition of consistency of constraints and explain how it captures the attacker's capability to distinguish processes based on their outputs.

To define consistency, we need the following ingredients:

- $\phi_\ell(c)$ and $\phi_r(c)$ denote the frames that are composed of the left and the right terms of the constraints respectively (in the same order).
- $\phi_{\text{LL}}^{\Gamma}$ denotes the frame that is composed of all low confidentiality nonces and keys in Γ, as well as all public encryption keys and verification keys in Γ. This intuitively corresponds

to the initial knowledge of the attacker.

- Let $\mathcal{E}_\Gamma$ be the set of all nonces occurring in Γ.
- Two ground substitutions σ, σ' are well-formed in Γ if they preserve the types for variables in Γ (i.e., $\Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$).

Definition 3 (Consistency)**.** *A set of constraints c is* consistent *in an environment Γ if for all substitutions σ,σ' well-typed in Γ the frames* `new` $\mathcal{E}_\Gamma.(\phi^\Gamma_{\mathrm{LL}} \cup \phi_\ell(c)\sigma)$ *and* `new` $\mathcal{E}_\Gamma.(\phi^\Gamma_{\mathrm{LL}} \cup \phi_r(c)\sigma')$ *are statically equivalent. We say that (c, Γ) is consistent if c is consistent in Γ and that a constraint set C is consistent in Γ if each element $(c, \Gamma) \in C$ is consistent.*

We define consistency of constraints in terms of static equivalence, as this notion exactly captures all capabilities of our attacker: to distinguish two processes, he can arbitrarily apply constructors and destructors on observed messages to create new terms, on which he can then perform equality tests or check the applicability of destructors. We require that this property holds for any well-typed substitutions, to soundly cover that fact that we do not know the content of variables statically, except for the information we get by typing. In Section 2.6.3 we introduce an algorithm to check consistency of constraints.

2.6 Main results

In this section, we state our two main soundness theorems, entailing trace equivalence by typing for the bounded and unbounded case, and we explain how to automatically check consistency.

2.6.1 Soundness of the type system

Our type system soundly enforces trace equivalence: if we can typecheck P and Q then P and Q are equivalent, provided that the corresponding constraint set is consistent.

Theorem 1 (Typing implies trace equivalence)**.** *For all P, Q, and C, for all Γ containing only keys, if $\Gamma \vdash P \sim Q \to C$ and C is consistent, then $P \approx_t Q$.*

To prove this theorem, we first show that typing is preserved by reduction, and guarantees that the same actions can be observed on both sides. More precisely, we show that if $\mathcal{P}$ and $\mathcal{Q}$ are multisets of processes that are pairwise typably equivalent (with consistent constraints), and if a reduction step with action α can be performed to reduce $\mathcal{P}$ into $\mathcal{P}'$, then $\mathcal{Q}$ can be reduced in one or several steps, with the same action α, to some multiset $\mathcal{Q}'$ such that the processes in $\mathcal{P}'$ and $\mathcal{Q}'$ are still typably equivalent (with consistent constraints). This is done by carefully examining all the possible typing rules used to type the processes in $\mathcal{P}$ and $\mathcal{Q}$. In addition we show that the frames of messages output when reducing $\mathcal{P}$ and $\mathcal{Q}$ are typably equivalent with consistent constraints; and that this entails their static equivalence.

This implies that if P and Q are typable with a consistent constraint, then for each trace of P, by induction on the length of the trace, there exists a trace of Q with the same sequence of actions,

and with a statically equivalent frame. That is to say $P \sqsubseteq_t Q$. Similarly we show $Q \sqsubseteq_t P$, and we thus have $P \approx_t Q$.

Since we do not have typing rules for replication, Theorem 1 only allows us to prove equivalence of protocols for a *finite* number of sessions. An arguably surprising result, however, is that, thanks to our infinite nonce types, we can prove equivalence for an *unbounded* number of sessions, as detailed in the next section.

2.6.2 Typing replicated processes

For more clarity, in this section, without loss of generality we consider that for each infinite nonce type $\tau_m^{l,\infty}$ appearing in the processes, the set of names $\mathcal{BN}$ contains an infinite number of fresh names $\{m_i \mid i \in \mathbb{N}\}$ which do not appear in the processes or environments. We similarly assume that for all the variables x appearing in the processes, the set $\mathcal{X}$ of all variables also contains fresh variables $\{x_i \mid i \in \mathbb{N}\}$ which do not appear in the processes or environments.

Intuitively, whenever we can typecheck a process of the form $\mathtt{new}\ n : \tau_n^{l,1}.\ \mathtt{new}\ m : \tau_m^{l,\infty}.\ P$, we can actually typecheck

$$\mathtt{new}\ n : \tau_n^{l,1}.\ (\mathtt{new}\ m_1 : \tau_{m_1}^{l,1}.P_1 \mid \ \ldots\ \mid \mathtt{new}\ m_k : \tau_{m_k}^{l,1}.P_k)$$

where in P_i, the nonce m has been replaced by m_i and variables x have been renamed to x_i.

Formally, we denote by $[\,t\,]_i^{\Gamma}$, the term t in which names n such that $\Gamma(n) = \tau_n^{l,\infty}$ for some l are replaced by n_i, and variables x are replaced by x_i.

Similarly, when a term is of type $[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$, it can be of type $[\![\tau_{m_i}^{l,1}\,;\,\tau_{p_i}^{l',1}]\!]$ for any i. The nonce type $\tau_m^{l,\infty}$ represents infinitely many nonces (one for each session). That is, for n sessions, the type $[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$ represents all $[\![\tau_{m_i}^{l,1}\,;\,\tau_{p_i}^{l',1}]\!]$. Formally, given a type T, we define its expansion to n sessions, denoted $[\,T\,]^n$, as follows.

$$\begin{aligned}
[\,l\,]^n &= l \\
[\,T * T'\,]^n &= [\,T\,]^n * [\,T'\,]^n \\
[\,T + T'\,]^n &= [\,T\,]^n + [\,T'\,]^n \\
\left[\,\mathrm{key}^l(T)\,\right]^n &= \mathrm{key}^l([\,T\,]^n) \\
[\,(T)_k\,]^n &= ([\,T\,]^n)_k \\
[\,\{T\}_k\,]^n &= \{[\,T\,]^n\}_k \\
[\,T \vee T'\,]^n &= [\,T\,]^n \vee [\,T'\,]^n \\
\left[\,[\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!]\,\right]^n &= [\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!] \\
\left[\,[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]\,\right]^n &= \textstyle\bigvee_{j=1}^{n} [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]
\end{aligned}$$

where $l, l' \in \{\mathtt{LL}, \mathtt{HH}, \mathtt{HL}\}$, $k \in \mathcal{K}$. Note that the size of the expanded type $[\,T\,]^n$ depends on n.

We need to adapt typing environments accordingly. For any typing environment Γ, we define its renaming for session i as:

$$\begin{aligned}[\,\Gamma\,]_i = \{x_i : T \mid \Gamma(x) = T\} \cup \{k : T \mid \Gamma(k) = T\} \\ \cup\ \{m : \tau_m^{l,1} \mid \Gamma(m) = \tau_m^{l,1}\} \\ \cup\ \{m_i : \tau_{m_i}^{l,1} \mid \Gamma(m) = \tau_m^{l,\infty}\}.\end{aligned}$$

and then its expansion to n sessions as

$$\begin{aligned}[\,\Gamma\,]_i^n = \{x_i : [\,T\,]^n \mid [\,\Gamma\,]_i(x_i) = T\} \cup \{k : [\,T\,]^n \mid [\,\Gamma\,]_i(k) = T\} \\ \cup\ \{m : \tau_m^{l,1} \mid [\,\Gamma\,]_i(m) = \tau_m^{l,1}\}.\end{aligned}$$

Note that in $[\,\Gamma\,]_i^n$, due to the expansion, the size of the types depends on n.

By construction, the environments contained in the constraints generated by typing do not contain union types. However, refinement types with infinite nonce types introduce union types when expanded. In order to recover environments without union types after expanding, which, as we will explain in the next subsection, is needed for our consistency checking procedure, we define $\mathrm{branches}([\,\Gamma\,]_i^n)$ as the set of all Γ', with the same domain as $[\,\Gamma\,]_i^n$, such that for all x, $\Gamma'(x)$ is not a union type, and either

- $[\,\Gamma\,]_i^n(x) = \Gamma'(x)$;
- or there exist types $T_1,\ldots,T_k,T'_1,\ldots,T'_{k'}$ such that

$$[\,\Gamma\,]_i^n(x) = T_1 \vee \ldots \vee T_k \vee \Gamma'(x) \vee T'_1 \vee \ldots \vee T'_{k'}$$

Finally, when typechecking two processes containing nonces with infinite nonce types, we collect constraints that represent families of constraints.

Given a set of constraints c, and an environment Γ, we define the renaming of c for session i in Γ as $[\,c\,]_i^\Gamma = \{[\,u\,]_i^\Gamma \sim [\,v\,]_i^\Gamma \mid u \sim v \in c\}$. This is propagated to constraint sets as follows: the renaming of C for session i is $[\,C\,]_i = \{([\,c\,]_i^\Gamma, [\,\Gamma\,]_i) \mid (c,\Gamma) \in C\}$ and its expansion to n sessions is $[\,C\,]_i^n = \{([\,c\,]_i^\Gamma, \Gamma') \mid \exists \Gamma.\ (c,\Gamma) \in C \ \wedge\ \Gamma' \in \mathrm{branches}([\,\Gamma\,]_i^n)\}$.

Again, note that the size of $[\,C\,]_i$ does not depend on the number of sessions considered, while the size of the types present in $[\,C\,]_i^n$ does. For example, for $C = \{(\{\mathtt{h}(x) \sim \mathtt{h}(x)\}, [x : [\![\tau_m^{\mathtt{HH},\infty}\,;\,\tau_p^{\mathtt{HH},\infty}]\!]])\}$, we have $[\,C\,]_i = \{(\{\mathtt{h}(x_i) \sim \mathtt{h}(x_i)\}, [x_i : [\![\tau_m^{\mathtt{HH},\infty}\,;\,\tau_p^{\mathtt{HH},\infty}]\!]])\}$ and $[\,C\,]_i^n = \{(\{\mathtt{h}(x_i) \sim \mathtt{h}(x_i)\}, [x_i : \bigvee_{j=1}^n [\![\tau_{m_j}^{\mathtt{HH},1}\,;\,\tau_{p_j}^{\mathtt{HH},1}]\!]])\}$.

Our type system is sound for replicated processes provided that the collected constraint sets are consistent, when instantiated with all possible instantiations of the nonces and keys.

Theorem 2. *Consider P, Q, P' ,Q', C, C', such that P, Q and P', Q' do not share any variable. Consider Γ, containing only keys and nonces with types of the form $\tau_n^{l,1}$.*

Assume that P and Q only bind nonces with infinite nonce types, i.e. *using* $\texttt{new}\ m : \tau_m^{l,\infty}$ *for some label l; while P' and Q' only bind nonces with finite types,* i.e. *using* $\texttt{new}\ m : \tau_m^{l,1}$.

Let us abbreviate by $\texttt{new}\ \overline{n}$ *the sequence of declarations of each nonce* $m \in \mathrm{dom}(\Gamma)$. *If*

- $\Gamma \vdash P \sim Q \to C$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- $C' \cup_\times (\cup_{\times 1 \le i \le n} [\, C \,]_i^n)$ *is consistent for all* n,

then $$\texttt{new}\ \overline{n}.\ ((!P) \mid P') \approx_t \texttt{new}\ \overline{n}.\ ((!Q) \mid Q').$$

Theorem 1 requires to check consistency of one constraint set. Theorem 2 now requires to check consistency of an infinite family of contraint sets. Instead of *deciding* consistency, we provide a procedure that checks a slightly stronger condition.

2.6.3 Procedure for consistency

Checking consistency of a set of constraints amounts to checking static equivalence of the corresponding frames. Our procedure follows the spirit of [AR00] for checking computational indistinguishability: we first open encryption, signatures and pairs as much as possible. Note that the type of a key indicates whether it is public or secret. The two resulting frames should have the same shape. Then, for unopened components, we simply need to check that they satisfy the same equalities.

From now on, we only consider constraint sets that can actually be generated when typing processes, as these are the only ones for which we need to check consistency.

Formally, the procedure `check_const` is described in Figure 2.1. It consists of four steps. First, we replace variables with refinements of finite nonce types by their left and right values. In particular a variable with a union type is not associated with a single value and thus cannot be replaced. This is why the branching operation needs to be performed when expanding environments containing refinements with types of the form $\tau_n^{l,\infty}$. Second, we recursively open the constraints as much as possible. Third, we check that the resulting constraints have the same shape. Finally, as soon as two constraints $M \sim M'$ and $N \sim N'$ are such that M, N are unifiable, we must have $M' = N'$, and conversely. The condition is slightly more involved, especially when the constraints contain variables of refined types with infinite nonce types.

Example 3. *Continuing Example 1, when typechecked with appropriate key types, the simplified model of Helios yields constraint sets containing notably the following two constraints.*

$$\{\ \texttt{aenc}(\langle 0, r_a\rangle, \texttt{pk}(k_s)) \sim \texttt{aenc}(\langle 1, r_a\rangle, \texttt{pk}(k_s)),$$
$$\texttt{aenc}(\langle 1, r_b\rangle, \texttt{pk}(k_s)) \sim \texttt{aenc}(\langle 0, r_b\rangle, \texttt{pk}(k_s))\ \}$$

$\texttt{step1}_\Gamma(c) := [\![c]\!]_{\sigma_F,\sigma'_F}$, with $F := \{x \in \mathrm{dom}(\Gamma) \mid \exists m,n,l,l'.\ \Gamma(x) = [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!]\}$ and σ_F, σ'_F defined by

- $\mathrm{dom}(\sigma_F) = \mathrm{dom}(\sigma'_F) = F$
- $\forall x \in F.\ \forall m,n,l,l'.[\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!] \Rightarrow \sigma_F(x) = m \ \wedge\ \sigma'_F(x) = n$

$\texttt{step2}_\Gamma(c)$ is recursively defined by, for all M, N, M', N':

- $\texttt{step2}_\Gamma(\{\langle M,N\rangle \sim \langle M',N'\rangle\} \cup c') := \texttt{step2}_\Gamma(\{M \sim M', N \sim N'\} \cup c')$
- For all $k \in \mathcal{K}$, if $\exists T.\Gamma(k) = \mathrm{key}^{\texttt{LL}}(T)$:
  - $\texttt{step2}_\Gamma(\{\texttt{enc}(M,k) \sim \texttt{enc}(M',k)\} \cup c, c') := \texttt{step2}_\Gamma(\{M \sim M'\} \cup c')$
  - $\texttt{step2}_\Gamma(\{\texttt{aenc}(M,\texttt{pk}(k)) \sim \texttt{aenc}(M',\texttt{pk}(k))\} \cup c, c') := \texttt{step2}_\Gamma(\{M \sim M'\} \cup c')$
  - $\texttt{step2}_\Gamma(\{\texttt{sign}(M,k) \sim \texttt{sign}(M',k)\} \cup c') := \texttt{step2}_\Gamma(\{M \sim M'\} \cup c')$
- For all $k \in \mathcal{K}$, if $\exists T.\Gamma(k) = \mathrm{key}^{\texttt{HH}}(T)$:

$$\texttt{step2}_\Gamma(\{\texttt{sign}(M,k) \sim \texttt{sign}(M',k)\} \cup c') := \{\texttt{sign}(M,k) \sim \texttt{sign}(M',k)\} \cup \texttt{step2}_\Gamma(\{M \sim M'\} \cup c')$$

- For all other terms M, N: $\texttt{step2}_\Gamma(\{M \sim N\} \cup c') := \{M \sim N\} \cup \texttt{step2}_\Gamma(c')$.

$\texttt{step3}_\Gamma(c) :=$ check that for all $M \sim N \in c$, M and N are both

- a key $k \in \mathcal{K}$ such that $\exists T.\Gamma(k) = \mathrm{key}^{\texttt{LL}}(T)$;
- nonces $m, n \in \mathcal{N}$ such that $\exists a \in \{1,\infty\}.\ \Gamma(n) = \tau_n^{\texttt{LL},a} \ \wedge\ \Gamma(m) = \tau_n^{\texttt{LL},a}$,
- or public keys, verification keys, or constants;
- or $\texttt{enc}(M',k)$, $\texttt{enc}(N',k)$ such that $\exists T.\Gamma(k) = \mathrm{key}^{\texttt{HH}}(T)$;
- or either $\texttt{h}(M')$, $\texttt{h}(N')$ or $\texttt{aenc}(M',\texttt{pk}(k))$, $\texttt{aenc}(N',\texttt{pk}(k))$, where $\exists T.\Gamma(k) = \mathrm{key}^{\texttt{HH}}(T)$; such that M' and N' contain directly under pairs some n with $\Gamma(n) = \texttt{HH}$ or k such that $\exists T.\Gamma(k) = \mathrm{key}^{\texttt{HH}}(T)$;
- or $\texttt{sign}(M',k)$, $\texttt{sign}(N',k)$ such that $\exists T.\Gamma(k) = \mathrm{key}^{\texttt{HH}}(T)$.

$\texttt{step4}_\Gamma(c) :=$ If for all $M \sim M'$ and $N \sim N' \in c$ where M, N are unifiable with a most general unifier μ, with $\forall x \in \mathrm{dom}(\mu).\exists l,l',m,p.\ (\Gamma(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]) \Rightarrow\ (x\mu \in \mathcal{X} \ \vee\ \exists i.\ x\mu = m_i)$, then we have $M'\alpha\theta = N'\alpha\theta$, where

$$\forall x \in \mathrm{dom}(\mu).\forall l,l',m,p,i.(\Gamma(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!] \ \wedge\ \mu(x) = m_i) \Rightarrow \theta(x) = p_i$$

and α is the restriction of μ to $\{x \in \mathrm{dom}(\mu) \mid \Gamma(x) = \texttt{LL} \ \wedge\ \mu(x) \in \mathcal{N}\}$; and if the symmetric condition for the case where M', N' are unifiable holds as well, then return `true`.

$\texttt{check\_const}(C) :=$ for all $(c,\Gamma) \in C$, let $c_1 := \texttt{step2}_\Gamma(\texttt{step1}_\Gamma(c))$ and check that $\texttt{step3}_\Gamma(c_1) = \texttt{true}$ and $\texttt{step4}_\Gamma(c_1) = \texttt{true}$.

Table 2.1: Procedure for checking consistency.

For simplicity, consider the set c containing only these two constraints, together with a typing environment Γ where r_a and r_b are respectively given types $\tau_{r_a}^{\mathtt{HH},1}$ and $\tau_{r_b}^{\mathtt{HH},1}$, and k_s is given type $\mathrm{key}^{\mathtt{HH}}(T)$ for some T.

The procedure $\mathtt{check\_const}(\{(c,\Gamma)\})$ *can detect that the constraint c is consistent and returns* $\mathtt{true}$*. Indeed, as c does not contain variables,* $\mathtt{step1}_\Gamma(c)$ *simply returns (c,Γ). Since c only contains messages encrypted with secret keys,* $\mathtt{step2}_\Gamma(c)$ *also leaves c unmodified.* $\mathtt{step3}_\Gamma(c)$ *then returns* $\mathtt{true}$*, since the messages appearing in c are messages asymmetrically encrypted with secret keys, which contain a secret nonce (r_a or r_b) directly under pairs. Finally* $\mathtt{step4}_\Gamma(c)$ *trivially returns* $\mathtt{true}$*, as the messages* $\mathtt{aenc}(\langle 0, r_a\rangle, \mathtt{pk}(k_s))$ *and* $\mathtt{aenc}(\langle 1, r_b\rangle, \mathtt{pk}(k_s))$ *cannot be unified, as well as the messages* $\mathtt{aenc}(\langle 1, r_a\rangle, \mathtt{pk}(k_s))$ *and* $\mathtt{aenc}(\langle 0, r_b\rangle, \mathtt{pk}(k_s))$*.*

Consider now the following set c', where encryption has not been randomised:

$$c' = \{\ \mathtt{aenc}(0, \mathtt{pk}(k_s)) \sim \mathtt{aenc}(1, \mathtt{pk}(k_s)),\\ \mathtt{aenc}(1, \mathtt{pk}(k_s)) \sim \mathtt{aenc}(0, \mathtt{pk}(k_s))\ \}$$

The procedure $\mathtt{check\_const}(\{(c',\Gamma)\})$ *returns* $\mathtt{false}$*. Indeed, contrary to the case of c,* $\mathtt{step3}_\Gamma(c')$ *fails, as the encrypted message do not contain a secret nonce. Actually, the corresponding frames are indeed not statically equivalent since the adversary can reconstruct the encryption of* 0 *and* 1 *with the key* $\mathtt{pk}(k_s)$ *(in his initial knowledge), and check for equality.*

For constraint sets without infinite nonce types, `check_const` entails consistency.

Theorem 3. *Let C be a set of constraints such that*

$$\forall (c,\Gamma) \in C.\ \forall l, l', m, p.\ \Gamma(x) \neq [\![\tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty}]\!].$$

If $\mathtt{check\_const}(C) = \mathtt{true}$*, then C is consistent.*

We prove this theorem by showing that, for each of the first two steps of the procedure, if $\mathtt{step}i_\Gamma(c)$ is consistent in Γ, then c is consistent in Γ. It then suffices to check the consistency of the constraint $\mathtt{step2}_\Gamma(\mathtt{step1}_\Gamma(c))$ in Γ. Provided that $\mathtt{step3}_\Gamma$ holds, we show that this constraint is saturated in the sense that any message obtained by the attacker by decomposing terms in the constraint already occurs in the constraint; and the constraint only contains messages which cannot be reconstructed by the attacker from the rest of the constraint. Using this property, we finally prove that the simple unification tests performed in `step4` are sufficient to ensure static equivalence of each side of the constraint for any well-typed instantiation of the variables.

As a direct consequence of Theorems 1 and 3, we now have a procedure to prove trace equivalence of processes without replication.

For proving trace equivalence of processes with replication, we need to check consistency of an infinite family of constraint sets, as prescribed by Theorem 2. As mentioned earlier, not only the number of constraints is unbounded, but the size of the type of some (replicated) variables is also unbounded (*i.e.* of the form $\bigvee_{j=1}^{n} [\![\tau_{m_j}^{l,1}\ ;\ \tau_{p_j}^{l',1}]\!]$). We use here two ingredients: we first show that it

is sufficient to apply our procedure to two constraints only. Second, we show that our procedure applied to variables with replicated types, *i.e.* nonce types of the form $\tau_n^{l,\infty}$ implies consistency of the corresponding constraints with types of unbounded size.

2.6.4 Two constraints suffice

Consistency of a constraint set C does not guarantee consistency of $\cup_{\times 1\leq i\leq n}[\, C\,]_i^n$. For example, consider

$$C = \{(\{\mathtt{h}(m) \sim \mathtt{h}(p)\}, [m : \tau_m^{\mathtt{HH},\infty}, p : \tau_p^{\mathtt{HH},1}])\}$$

which can be obtained when typing

$$\begin{array}{l}\mathtt{new}\ m : \tau_m^{\mathtt{HH},\infty}.\ \mathtt{new}\ p : \tau_p^{\mathtt{HH},1}.\ \mathtt{out}(\mathtt{h}(m)) \sim \\ \mathtt{new}\ m : \tau_m^{\mathtt{HH},\infty}.\ \mathtt{new}\ p : \tau_p^{\mathtt{HH},1}.\ \mathtt{out}(\mathtt{h}(p)).\end{array}$$

C is consistent: since m, p are secret, the attacker cannot distinguish between their hashes. However $\cup_{\times 1\leq i\leq n}[\, C\,]_i^n$ contains (together with some environment):

$$\{\mathtt{h}(m_1) \sim \mathtt{h}(p), \mathtt{h}(m_2) \sim \mathtt{h}(p), \ldots, \mathtt{h}(m_n) \sim \mathtt{h}(p)\}$$

which is not, since the attacker can notice that the value on the right is always the same, while the value on the left is not.

Note however that the inconsistency of $\cup_{\times 1\leq i\leq n}[\, C\,]_i^n$ would have been discovered when checking the consistency of two copies of the constraint set only. Indeed, $[\, C\,]_1^n \cup_\times [\, C\,]_2^n$ contains (together with some environment):

$$\{\mathtt{h}(m_1) \sim \mathtt{h}(p), \mathtt{h}(m_2) \sim \mathtt{h}(p)\}$$

which is already inconsistent, for the same reason.

Actually, checking consistency (with our procedure) of two constraints $[\, C\,]_1^n$ and $[\, C\,]_2^n$ entails consistency of $\cup_{\times 1\leq i\leq n}[\, C\,]_i^n$. Note that this does not mean that consistency of $[\, C\,]_1^n$ and $[\, C\,]_2^n$ implies consistency of $\cup_{\times 1\leq i\leq n}[\, C\,]_i^n$. Instead, our procedure ensures a stronger property, for which two constraints suffice.

Theorem 4. *Let C and C' be two constraint sets, which do not contain any common variables. For all $n \in \mathbb{N}$,*

$$\begin{array}{l}\mathtt{check\_const}([\, C\,]_1^n \cup_\times [\, C\,]_2^n \cup_\times [\, C'\,]_1^n) = \mathtt{true} \ \Rightarrow \\ \quad \mathtt{check\_const}((\cup_{\times 1\leq i\leq n}[\, C\,]_i^n) \cup_\times [\, C'\,]_1^n) = \mathtt{true}.\end{array}$$

To prove Theorem 4, we first (easily) show that if

$$\mathtt{check\_const}([\, C\,]_1^n \cup_\times [\, C\,]_2^n \cup_\times [\, C'\,]_1^n) = \mathtt{true},$$

then the first three steps of the procedure `check_const` can be successfully applied to each element of $(\cup_{\times 1\leq i\leq n}[\, C\,]_i^n) \cup_\times [\, C'\,]_1^n$. However the case of the fourth step is more intricate. When applying the procedure `check_const` to an element of the constraint set $(\cup_{\times 1\leq i\leq n}[\, C\,]_i^n) \cup_\times [\, C'\,]_1^n$,

if `step4` fails, then the constraint contains an inconsistency, *i.e.* elements $M \sim M'$ and $N \sim N'$ for which the unification condition from `step4` does not hold. Then we show that we can find a similar inconsistency when considering only the first two constraint sets, *i.e.* in $[\, C \,]_1^n \cup_\times [\, C \,]_2^n \cup_\times [\, C' \,]_1^n$. This is done by reindexing the nonces and variables. The proof actually requires a careful examination of the structure of the constraint set $(\cup_{\times 1 \leq i \leq n} [\, C \,]_i^n) \cup_\times [\, C' \,]_1^n$, to establish this reindexing.

2.6.5 Reducing the size of types

The procedure `check_const` applied to replicated types implies consistency of corresponding constraints with unbounded types.

Theorem 5. *Let C be a constraint set. Then for all i,*

$$\texttt{check\_const}([\, C \,]_i) = \texttt{true} \;\Rightarrow\; \forall n \geq 1. \texttt{check\_const}([\, C \,]_i^n) = \texttt{true}$$

Again here, it is rather easy to show that if $\texttt{check\_const}([\, C \,]_i) = \texttt{true}$ then the first three steps of the procedure `check_const` can successfully be applied to each element of $[\, C \,]_i^n$. The case of `step4` is more involved. The property holds thanks to the condition on the most general unifier expressed in `step4`. Intuitively, this condition is written in such a way that if, when applying `step4` to an element of $[\, C \,]_i^n$, two messages can be unified, then the corresponding messages (with replicated types) in $[\, C \,]_i$ can be unified with a most general unifier μ satisfying the condition. The proof uses this idea to show that if `step4` succeeds on all elements of $[\, C \,]_i$, then it also succeeds on the elements of $[\, C \,]_i^n$.

2.6.6 Checking the consistency of the infinite constraint

Theorems 2, 4, and 5 provide a sound procedure for checking trace equivalence of processes with and without replication.

Theorem 6. *Let C, and C' be two constraint sets without any common variable.*

$$\texttt{check\_const}([\, C \,]_1 \cup_\times [\, C \,]_2 \cup_\times [\, C' \,]_1) = \texttt{true} \;\Rightarrow\; \forall n.\, [\, C' \,]_1^n \cup_\times (\cup_{\times 1 \leq i \leq n} [\, C \,]_i^n) \textit{ is consistent.}$$

All detailed proofs are available online [CGLM17b].

2.7 Experimental results

We have implemented a prototype type-checker TypeEq and applied it on various examples briefly described below.

Symmetric key protocols. For the sake of comparison, we consider 5 symmetric key protocols taken from the benchmark of [CDD17], and described in [CJ97]: Denning-Sacco, Wide Mouth

Frog, Needham-Schroeder, Yahalom-Lowe, and Otway-Rees. All these protocols aim at exchanging a key k. We prove strong secrecy of the key, as defined in [Aba00], i.e., $P(k_1) \approx_t P(k_2)$ where k_1 and k_2 are public names. Intuitively, an attacker should not be able to tell which key is used even if he knows the two possible values in advance. For some of the protocols, we truncated the last step, when it consists in using the exchanged key for encryption, since our framework currently covers only encryption with long-term (fixed) keys.

Asymmetric key protocols. In addition to the symmetric key protocols, we consider the well-known Needham-Schroeder-Lowe (NSL) protocol [Low96] and we again prove strong secrecy of the nonce sent by the receiver (Bob).

Helios. We model the Helios protocol for two honest voters and infinitely many dishonest ones, as informally described in Section 2.2. The corresponding process includes a non trivial else branch, used to express the weeding phase [CS11], where dishonest ballots equal to some honest one are discarded. As emphasised in Section 2.2, Helios is secure only if honest voters vote at most once. Therefore the protocol includes non replicated processes (for voters) as well as a replicated process (to handle dishonest ballots).

All our experiments have been run on a single Intel Xeon E5-2687Wv3 3.10GHz core, with 378GB of RAM (shared with the 19 other cores). All corresponding files can be found online at [CGLM17c].

2.7.1 Bounded number of sessions

We first compare our tool with tools designed for a bounded number of sessions: SPEC [DT10], APTE (and its APTE-POR variant) [Che14, BDH15], Akiss [CCK12], or SAT-Equiv [CDD17]. The protocol models may slightly differ due to the subtleties of each tool. For example, several of these tools require *simple* processes where each sub-process emits on a distinct channel. We do not need such an assumption. In addition, SAT-Equiv only covers symmetric encryption and therefore could not be applied to Helios or NSL. SAT-Equiv further assumes protocols to be well-typed, which sometimes requires to tag protocols. Since we consider only untagged versions (following the original description of each protocol), SAT-Equiv failed to prove the Otway-Rees protocol. Moreover, Helios involves non-trivial else branches, which are only supported by APTE.

The number of sessions we consider denotes the number of processes in parallel in each scenario. For symmetric key protocols, we start with a simple scenario with only two honest participants A, B and a honest server S (3 sessions). We consider increasingly more complex scenarios (6, 7, 10, 12, and 14 sessions) featuring a dishonest agent C. In the complete scenario (14 sessions) each agent among A, B (and C) runs the protocol once as the initiator, and once as the responder with each other agent (A, B, C). In the case of NSL, we similarly consider a scenario with two honest agents A, B running the protocol once (2 sessions), and two scenarios with an additional dishonest agent C, up to the complete scenario (8 sessions) where each agent runs NSL once as initiator, once as responder, with each agent. For Helios, we consider 2 honest voters, and one dishonest voter only, as well as a ballot box. The corresponding results are reported in Figure 2.9. We write TO for Time Out (12 hours), MO for Memory Out (more than 64 GB of RAM), SO for

Protocols (# sessions)		Akiss	APTE	APTE-POR	Spec	Sat-Eq	TypeEq
Denning - Sacco	3	0.08s	0.32s	0.02s	9s	0.09s	0.002s
	6	3.9s	TO	1.6s	191m	0.3s	0.003s
	7	29s		3.6s	TO	0.8s	0.004s
	10	SO		12m		1.8s	0.004s
	12			TO		3.4s	0.005s
	14					5s	0.006s
Wide Mouth Frog	3	0.03s	0.05s	0.009s	8s	0.06s	0.002s
	6	0.4s	28m	0.4s	52m	0.2s	0.003s
	7	1.4s	TO	1.9s	MO	2.3s	0.003s
	10	46s		5m31s		5s	0.004s
	12	71m		TO		1m	0.005s
	14	TO				4m20s	0.006s
Needham - Schroeder	3	0.1s	0.4s	0.02s	52s	0.5s	0.003s
	6	20s	TO	4s	MO	4s	0.003s
	7	2m		8m		36s	0.003s
	10	SO		TO		1m50s	0.005s
	12					4m47s	0.005s
	14					11m	0.007s
Yahalom - Lowe	3	0.16s	3.6s	0.03s	6s	1.4s	0.003s
	6	33s	TO	44s	132m	1m	0.004s
	7	11m		36m	MO	17m	0.004s
	10	SO		TO		63m	0.009s
	12					TO	0.04s
	14						0.05s
Otway-Rees	3	2m12s	BUG	1.7s	27m	x	0.004s
	6	TO		SO	MO		0.011s
	7						0.012s
	10						0.02s
	12						0.03s
	14						0.1s
Needham-Schroeder-Lowe	2	0.1s	4s	0.06s	31s	x	0.003s
	4	2m	BUG	BUG	MO		0.003s
	8	TO					0.007s
Helios	3	x	TO	BUG	x	x	0.002s

Figure 2.9: Experimental results for the bounded case

Protocols	ProVerif	TypeEq
Helios	x	0.003s
Denning-Sacco	0.05s	0.05s
Needham-Schroeder-Lowe	0.08s	0.09s

Figure 2.10: Experimental results for unbounded numbers of sessions

Stack Overflow, BUG in the case of APTE, when the proof failed due to bugs in the tool, and x when the tool could not handle the protocol for the reasons discussed previously. In all cases, our tool is almost instantaneous and outperforms by orders of magnitude the competitors.

2.7.2 Unbounded numbers of sessions

We then compare our type-checker with ProVerif [BAF08], for an unbounded number of sessions, on three examples: Helios, Denning-Sacco, and NSL. As expected, ProVerif cannot prove Helios secure since it cannot express that voters vote only once. This may sound surprising, since proofs of Helios in ProVerif already exist (e.g. [CS11, ACK16]). Interestingly, these models actually implicitly assume a reliable channel between honest voters and the voting server: whenever a voter votes, she first sends her vote to the voting server on a secure channel, before letting the attacker see it. This model prevents an attacker from reading and blocking a message, while this can be easily done in practice (by breaking the connection). We also failed to prove (automatically) Helios in Tamarin [BDS15]. The reason is that the weeding procedure makes Tamarin enter a loop where it cannot detect that, as soon as a ballot is not weed, it has been forged by the adversary.

For the sake of comparison, we run both tools (ProVerif and TypeEq) on a symmetric protocol (Denning-Sacco) and an asymmetric protocol (Needham-Schroeder-Lowe). The execution times are very similar. The corresponding results are reported in Figure 2.10.

2.8 Conclusion

We presented a novel type system for verifying trace equivalence in security protocols. It can be applied to various protocols, with support for else branches, standard cryptographic primitives, as well as a bounded and an unbounded number of sessions. We believe that our prototype implementation demonstrates that this approach is promising and opens the way to the development of an efficient technique for proving equivalence properties in even larger classes of protocols.

Several interesting problems remain to be studied. For example, a limitation of ProVerif is that it cannot properly handle global states. We plan to explore this case by enriching our types to express the fact that an event is “consumed”. Also, for the moment, our type system only applies to protocols P, Q that have the same structure. One advantage of a type system is its modularity: it is relatively easy to add a few rules without redoing the whole proof. We plan to add rules to cover protocols with different structures (e.g. when branches are swapped). Another direction is the

treatment of primitives with algebraic properties (e.g. Exclusive Or, or homomorphic encryption). It seems possible to extend the type system and discharge the difficulty to the consistency of the constraints, which seems easier to handle (since this captures the static case). Finally, our type system is sound w.r.t. equivalence in a symbolic model. An interesting question is whether it also entails computational indistinguishability. Again, we expect that an advantage of our type system is the possibility to discharge most of the difficulty to the constraints.

CHAPTER 3

Extending the Type System to Branching Protocols

Abstract

Recently, many tools have been proposed for automatically analysing, in symbolic models, equivalence of security protocols. Equivalence is a property needed to state privacy properties or game-based properties like strong secrecy. Tools for a bounded number of sessions can decide equivalence but typically suffer from efficiency issues. Tools for an unbounded number of sessions like Tamarin or ProVerif prove a stronger notion of equivalence (diff-equivalence) that does not properly handle protocols with else branches.
Building upon a recent approach, we propose a type system for reasoning about branching protocols and dynamic keys. We prove our type system to entail equivalence, for all the standard primitives. Our type system has been implemented and shows a significant speedup compared to the tools for a bounded number of sessions, and compares similarly to ProVerif for an unbounded number of sessions. Moreover, we can also prove security of protocols that require a mix of bounded and unbounded number of sessions, which ProVerif cannot properly handle.

This chapter presents the second result of a collaboration with Véronique Cortier, Joseph Lallemand and Matteo Maffei and was published at the 7th International Conference on Principles of Security and Trust (POST'18) under the title "Equivalence Properties by Typing in Cryptographic Branching Protocols" [CGLM18a]. I contributed to the design of the type system and the evaluation and am responsible for the reference implementation. The design of consistency procedure and the formal proofs were done mostly by Joseph Lallemand.

3.1 Introduction

Formal methods provide a rigorous and convenient framework for analysing security protocols. In particular, mature push-button analysis tools have emerged and have been successfully applied to many protocols from the literature in the context of *trace properties* such as authentication or confidentiality. These tools employ a variety of analysis techniques, such as model checking (e.g., Avispa [ABB+05] and Scyther [Cre08]), Horn clause resolution (e.g., ProVerif [Bla01]), term rewriting (e.g., Scyther [Cre08] and Tamarin [MSCB13]), and type systems [GJ03, BFM04, BFM05, BFM07, BBF+11, FM11, BCEM11, EM13, BCEM13, BHM14, BCEM15].

In the recent years, attention has been given also to equivalence properties, which are crucial to model privacy properties such as vote privacy [BHM08, DKR09], unlikability [ACRR10], or anonymity [BMU08]. For example, consider an authentication protocol P_{pass} embedded in a biometric passport. P_{pass} preserves anonymity of passport holders if an attacker cannot distinguish an execution with Alice from an execution with Bob. This can be expressed by the equivalence $P_{pass}(Alice) \approx_t P_{pass}(Bob)$. Equivalence is also used to express properties closer to cryptographic games like strong secrecy.

Two main classes of tools have been developed for equivalence. First, in the case of an unbounded number of sessions (when the protocol is executed arbitrarily many times), equivalence is undecidable. Instead, the tools ProVerif [Bla01, BAF08] and Tamarin [MSCB13, BDS15] try to prove a stronger property, namely diff-equivalence, that may be too strong e.g. in the context of voting. Tamarin covers a larger class of protocols but may require some guidance from the user. Maude-NPA [EMM06, SEMM14] also proves diff-equivalence but may have non-termination issues. Another class of tools aim at deciding equivalence, for bounded number of sessions. This is the case in particular of SPEC [DT10], APTE [Che14], Akiss [CCK12], and SatEquiv [CDD17]. SPEC, APTE, and Akiss suffer from efficiency issues and can typically not handle more than 3-4 sessions. SatEquiv is much more efficient but is limited to symmetric encryption and requires protocols to be well-typed, which often assumes some additional tagging of the protocol.

Our contribution. Following the approach of [CGLM17a], we propose a novel technique for proving equivalence properties for a bounded number of sessions as well as an unbounded number of sessions (or a mix of both), based on typing. [CGLM17a] proposes a first type system that entails trace equivalence $P \approx_t Q$, provided protocols use fixed (long-term) keys, identical in P and Q. In this paper, we target a larger class of protocols, that includes in particular key-exchange protocols and protocols whose security relies on branching on the secret. This is the case e.g. of the private authentication protocol [AF04], where agent B returns a true answer to A, encrypted with A's public key if A is one of his friends, and sends a decoy message (encrypted with a dummy key) otherwise.

We devise a new type system for reasoning about keys. In particular, we introduce bikeys to cover behaviours where keys in P differ from the keys in Q. We design new typing rules to reason about protocols that may branch differently (in P and Q), depending on the input. Following the approach of [CGLM17a], our type system collects sent messages into constraints that are required to be consistent. Intuitively, the type system guarantees that any execution of P can be matched by an execution of Q, while consistency imposes that the resulting sequences of messages are

indistinguishable for an attacker. We had to entirely revisit the approach of [CGLM17a] and prove a finer invariant in order to cope with the case where keys are used as variables. Specifically, most of the rules for encryption, signature, and decryption had to be adapted to accommodate the flexible usage of keys. For messages, we had to modify the rules for keys and encryption, in order to encrypt messages with keys of different type (bi-key type), instead of only fixed keys. We show that our type system entails equivalence for the standard notion of trace equivalence [CCD13] and we devise a procedure for proving consistency. This yields an efficient approach for *proving* equivalence of protocols for a bounded and an unbounded number of sessions (or a combination of both).

We implemented a prototype of our type-checker that we evaluate on a set of examples, that includes private authentication, the BAC protocol (of the biometric passport), as well as Helios together with the setup phase. Our tool requires a light type annotation that specifies which keys and names are likely to be secret or public and the form of the messages encrypted by a given key. This can be easily inferred from the structure of the protocol. Our type-checker outperforms even the most efficient existing tools for a bounded number of sessions by two (for examples with few processes) to three (for examples with more processes) orders of magnitude. Note however that these tools *decide* equivalence while our type system is incomplete. In the case of an unbounded number of sessions, on our examples, the performance is comparable to ProVerif, one of the most popular tools. We consider in particular vote privacy in the Helios protocol, in the case of a dishonest ballot board, with no revote (as the protocol is insecure otherwise). ProVerif fails to handle this case as it cannot (faithfully) consider a mix of bounded and unbounded number of sessions. Compared to [CGLM17a], our analysis includes the setup phase (where voters receive the election key), which could not be considered before.

The technical details and proofs omitted due to space constraints are available in the companion technical report [CGLM18b].

3.2 High-level description

3.2.1 Background

Trace equivalence of two processes is a property that guarantees that an attacker observing the execution of either of the two processes cannot decide which one it is. Previous work [CGLM17a] has shown how trace equivalence can be proved statically using a type system combined with a constraint checking procedure. The type system consists of typing rules of the form $\Gamma \vdash P \sim Q \to C$, meaning that in an environment Γ two processes P and Q are equivalent if the produced set of constraints C, encoding the attacker observables, is consistent.

The typing environment Γ is a mapping from nonces, keys, and variables to types. Nonces are assigned security labels with a confidentiality and an integrity component, e.g. $\mathtt{HL}$ for high confidentiality and low integrity. Key types are of the form $\mathrm{key}^l(T)$ where l is the security label of the key and T is the type of the payload. Key types are crucial to convey typing information from one process to another one. Normally, we cannot make any assumptions about values received from the network – they might possibly originate from the attacker. If we however successfully

decrypt a message using a secret symmetric key, we know that the result is of the key's payload type. This is enforced on the sender side, whenever outputting an encryption.

A core assumption of virtually any efficient static analysis for equivalence is uniform execution, meaning that the two processes of interest always take the same branch in a branching statement. For instance, this means that all decryptions must always succeed or fail equally in the two processes. For this reason, previous work introduced a restriction to allow only encryption and decryption with keys whose equality could be statically proved.

3.2.2 Limitation

There are however protocols that require non-uniform execution for a proof of trace equivalence, e.g., the private authentication protocol [AF04]. The protocol aims at authenticating B to A, anonymously w.r.t. other agents. More specifically, agent B may refuse to communicate with agent A but a third agent D should not learn whether B declines communication with A or not. The protocol can be informally described as follows, where $\mathtt{pk}(k)$ denotes the public key associated to key k, and $\mathtt{aenc}(M, \mathtt{pk}(k))$ denotes the asymmetric encryption of message M with this public key.

$$
\begin{array}{ll}
A \to B: & \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, \mathtt{pk}(k_b)) \\
B \to A: & \begin{cases} \mathtt{aenc}(\langle N_a, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) & \text{if } B \text{ accepts } A\text{'s request} \\ \mathtt{aenc}(N_b, \mathtt{pk}(k)) & \text{if } B \text{ declines } A\text{'s request} \end{cases}
\end{array}
$$

If B declines to communicate with A, he sends a decoy message $\mathtt{aenc}(N_b, \mathtt{pk}(k))$ where $\mathtt{pk}(k)$ is a decoy key (no one knows the private key k).

3.2.3 Encrypting with different keys

Let $P_a(k_a, \mathtt{pk}(k_b))$ model agent A willing to talk with B, and $P_b(k_b, \mathtt{pk}(k_a))$ model agent B willing to talk with A (and declining requests from other agents). We model the protocol as:

$$
\begin{array}{rcl}
P_a(k_a, pk_b) & = & \mathtt{new}\ N_a.\mathtt{out}(\mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b)).\ \mathtt{in}(z) \\
P_b(k_b, pk_a) & = & \mathtt{new}\ N_b.\ \mathtt{in}(x). \\
 & & \mathtt{let}\ y = \mathtt{adec}(x, k_b)\ \mathtt{in}\ \mathtt{let}\ y_1 = \pi_1(y)\ \mathtt{in}\ \mathtt{let}\ y_2 = \pi_2(y)\ \mathtt{in} \\
 & & \quad \mathtt{if}\ y_2 = pk_a\ \mathtt{then} \\
 & & \quad\quad \mathtt{out}(\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, pk_a)) \\
 & & \quad \mathtt{else}\ \mathtt{out}(\mathtt{aenc}(N_b, \mathtt{pk}(k)))
\end{array}
$$

where $\mathtt{adec}(M, k)$ denotes asymmetric decryption of message M with private key k. We model anonymity as the following equivalence, intuitively stating that an attacker should not be able to tell whether B accepts requests from the agent A or C:

$$P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a)) \approx_t P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_c))$$

We now show how we can type the protocol in order to show trace equivalence. The initiator P_a is trivially executing uniformly, since it does not contain any branching operations. We hence focus on typing the responder P_b.

$$
\begin{array}{rcll}
\Gamma(k_b, k_b) & = & \text{key}^{\texttt{HH}}(\texttt{HL} * \texttt{LL}) & \text{initial message uses same key on both sides} \\
\Gamma(k_a, k) & = & \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication succeeded on the left, failed on the right} \\
\Gamma(k, k_c) & = & \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication succeeded on the right, failed on the left} \\
\Gamma(k_a, k_c) & = & \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication succeeded on both sides} \\
\Gamma(k, k) & = & \text{key}^{\texttt{HH}}(\texttt{HL}) & \text{authentication failed on both sides}
\end{array}
$$

Figure 3.1: Key types for the private authentication protocol

The beginning of the responder protocol can be typed using standard techniques. Then however, we perform the test $y_2 = \texttt{pk}(k_a)$ on the left side and $y_2 = \texttt{pk}(k_c)$ on the right side. Since we cannot statically determine the result of the two equality checks – and thus guarantee uniform execution – we have to typecheck the four possible combinations of `then` and `else` branches. This means we have to typecheck outputs of encryptions that use different keys on the left and the right side.

To deal with this we do not assign types to single keys, but rather to pairs of keys (k, k') – which we call *bikeys* – where k is the key used in the left process and k' is the key used in the right process. The key types used for typing are presented in Fig. 3.1.

As an example, we consider the combination of the `then` branch on the left with the `else` branch on the right. This combination occurs when A is successfully authenticated on the left side, while being rejected on the right side. We then have to typecheck B's positive answer together with the decoy message: $\Gamma \vdash \texttt{aenc}(\langle y_1, \langle N_b, \texttt{pk}(k_b)\rangle\rangle, \texttt{pk}(k_a)) \sim \texttt{aenc}(N_b, \texttt{pk}(k)) : \texttt{LL}$. For this we need the type for the bikey (k_a, k).

3.2.4 Decrypting non-uniformly

When decrypting a ciphertext that was potentially generated using two different keys on the left and the right side, we have to take all possibilities into account. Consider the following extension of the process P_a where agent A decrypts B's message.

$$
\begin{array}{rcl}
P_a(k_a, pk_b) & = & \texttt{new}\ N_a.\texttt{out}(\texttt{aenc}(\langle N_a, \texttt{pk}(k_a)\rangle, pk_b)).\ \texttt{in}(z). \\
& & \texttt{let}\ z' = \texttt{adec}(z, k_a)\ \texttt{in}\ \texttt{out}(1) \\
& & \texttt{else}\ \texttt{out}(0)
\end{array}
$$

In the decryption, there are the following possible cases:

- The message is a valid encryption supplied by the attacker (using the public key $\texttt{pk}(k_a)$), so we check the `then` branch on both sides with $\Gamma(z') = \texttt{LL}$.
- The message is not a valid encryption supplied by the attacker so we check the `else` branch on both sides.

- The message is a valid response from B. The keys used on the left and the right are then one of the four possible combinations $(k_a, k), (k_a, k_c), (k, k_c)$ and (k, k).
  - In the first two cases the decryption will succeed on the left and fail on the right. We hence check the `then` branch on the left with $\Gamma(z') = \texttt{HL}$ with the `else` branch on the right. If the type $\Gamma(k_a, k)$ were different from $\Gamma(k_a, k_c)$, we would check this combination twice, using the two different payload types.
  - In the remaining two cases the decryption will fail on both sides. We hence would have to check the two `else` branches (which however we already did).

While checking the `then` branch together with the `else` branch, we have to check $\Gamma \vdash 1 \sim 0 : \texttt{LL}$, which rightly fails, as the protocol does not guarantee trace equivalence.

3.3 Model

In symbolic models, security protocols are typically modelled as processes of a process algebra, such as the applied pi-calculus [AF01]. We present here a calculus used in [CGLM17a] and inspired from the calculus underlying the ProVerif tool [Bla16]. This section is mostly an excerpt of [CGLM17a], recalled here for the sake of completeness, and illustrated with the private authentication protocol.

3.3.1 Terms

Messages are modelled as terms. We assume an infinite set of names $\mathcal{N}$ for nonces, further partitioned into the set $\mathcal{FN}$ of free nonces (created by the attacker) and the set $\mathcal{BN}$ of bound nonces (created by the protocol parties), an infinite set of names $\mathcal{K}$ for keys similarly split into $\mathcal{FK}$ and $\mathcal{BK}$, and an infinite set of variables $\mathcal{V}$. Cryptographic primitives are modelled through a *signature* $\mathcal{F}$, that is, a set of function symbols, given with their arity (*i.e.* the number of arguments). Here, we consider the following signature:

$$\mathcal{F}_c = \{\texttt{pk}, \texttt{vk}, \texttt{enc}, \texttt{aenc}, \texttt{sign}, \langle\cdot,\cdot\rangle, \texttt{h}\}$$

that models respectively public and verification key, symmetric and asymmetric encryption, concatenation and hash. The companion primitives (symmetric and asymmetric decryption, signature check, and projections) are represented by the following signature:

$$\mathcal{F}_d = \{\texttt{dec}, \texttt{adec}, \texttt{checksign}, \pi_1, \pi_2\}$$

We also consider a set $\mathcal{C}$ of (public) constants (used as agent names for instance). Given a signature $\mathcal{F}$, a set of names $\mathcal{N}$, and a set of variables $\mathcal{V}$, the set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{V}, \mathcal{N})$ is the set inductively defined by applying functions to variables in $\mathcal{V}$ and names in $\mathcal{N}$. We denote by $\mathrm{names}(t)$ (resp. $\mathrm{vars}(t)$) the set of names (resp. variables) occurring in t. A term is *ground* if it does not contain variables.

We consider the set $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ of *cryptographic terms*, simply called *terms*. *Messages* are terms with constructors from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$. We assume the set of variables to be split into two subsets $\mathcal{V} = \mathcal{X} \uplus \mathcal{AX}$ where $\mathcal{X}$ are variables used in processes while $\mathcal{AX}$ are variables used to store messages. An *attacker term* is a term from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{AX}, \mathcal{FN} \cup \mathcal{FK})$. In particular, an attacker term cannot use nonces and keys created by the protocol's parties.

A *substitution* $\sigma = \{M_1/x_1, \ldots, M_k/x_k\}$ is a mapping from variables $x_1, \ldots, x_k \in \mathcal{V}$ to messages $M_1, \ldots, M_k$. We let $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_k\}$. We say that σ is ground if all messages $M_1, \ldots, M_k$ are ground. We let $\mathrm{names}(\sigma) = \bigcup_{1 \le i \le k} \mathrm{names}(M_i)$. The application of a substitution σ to a term t is denoted $t\sigma$ and is defined as usual.

The *evaluation* of a term t, denoted $t\downarrow$, corresponds to the bottom-up application of the cryptographic primitives and is recursively defined as follows.

$$
\begin{array}{rl}
u\downarrow = u & \text{if } u \in \mathcal{N} \cup \mathcal{V} \cup \mathcal{K} \cup \mathcal{C} \\
\mathtt{pk}(t)\downarrow = \mathtt{pk}(t\downarrow) & \text{if } t\downarrow \in \mathcal{K} \\
\mathtt{vk}(t)\downarrow = \mathtt{vk}(t\downarrow) & \text{if } t\downarrow \in \mathcal{K} \\
\mathtt{h}(t)\downarrow = \mathtt{h}(t\downarrow) & \text{if } t\downarrow \neq \bot \\
\langle t_1, t_2\rangle\downarrow = \langle t_1\downarrow, t_2\downarrow\rangle & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \neq \bot \\
\mathtt{enc}(t_1, t_2)\downarrow = \mathtt{enc}(t_1\downarrow, t_2\downarrow) & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \in \mathcal{K} \\
\mathtt{sign}(t_1, t_2)\downarrow = \mathtt{sign}(t_1\downarrow, t_2\downarrow) & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow \in \mathcal{K} \\
\mathtt{aenc}(t_1, t_2)\downarrow = \mathtt{aenc}(t_1\downarrow, t_2\downarrow) & \text{if } t_1\downarrow \neq \bot \text{ and } t_2\downarrow = \mathtt{pk}(k) \\
\qquad \text{for some } k \in \mathcal{K} &
\end{array}
$$

$$
\begin{array}{rl}
\pi_1(t)\downarrow = t_1 & \text{if } t\downarrow = \langle t_1, t_2\rangle \\
\pi_2(t)\downarrow = t_2 & \text{if } t\downarrow = \langle t_1, t_2\rangle \\
\mathtt{dec}(t_1, t_2)\downarrow = t_3 & \text{if } t_1\downarrow = \mathtt{enc}(t_3, t_4) \text{ and } t_4 = t_2\downarrow \\
\mathtt{adec}(t_1, t_2)\downarrow = t_3 & \text{if } t_1\downarrow = \mathtt{aenc}(t_3, \mathtt{pk}(t_4)) \text{ and } t_4 = t_2\downarrow \\
\mathtt{checksign}(t_1, t_2)\downarrow = t_3 & \text{if } t_1\downarrow = \mathtt{sign}(t_3, t_4) \text{ and } t_2\downarrow = \mathtt{vk}(t_4) \\
t\downarrow = \bot & \text{otherwise}
\end{array}
$$

Note that the evaluation of term t succeeds only if the underlying keys are atomic and always returns a message or $\bot$. For example we have $\pi_1(\langle a, b\rangle)\downarrow = a$, while $\mathtt{dec}(\mathtt{enc}(a, \langle b, b\rangle), \langle b, b\rangle)\downarrow = \bot$, because the key is non atomic. We write $t =_\downarrow t'$ if $t\downarrow = t'\downarrow$.

3.3.2 Processes

Security protocols describe how messages should be exchanged between participants. We model them through a process algebra, whose syntax is displayed in Fig. 3.2. We identify processes up to α-renaming, *i.e.*, avoiding substitution of bound names and variables, which are defined as usual. Furthermore, we assume that all bound names, keys, and variables in the process are distinct.

A *configuration* of the system is a tuple $(\mathcal{P}; \phi; \sigma)$ where:

- $\mathcal{P}$ is a multiset of processes that represents the current active processes;

Destructors used in processes:

$$d ::= \mathtt{dec}(x,t) \mid \mathtt{adec}(x,t) \mid \mathtt{checksign}(x,t') \mid \pi_1(x) \mid \pi_2(x)$$

where $x \in \mathcal{X}, t \in \mathcal{K} \cup \mathcal{X}, t' \in \{\mathtt{vk}(k) | k \in \mathcal{K}\} \cup \mathcal{X}$.

Processes:

$$\begin{aligned} P,Q \quad ::= \quad & 0 \mid \mathtt{new}\ n.P \mid \mathtt{out}(M).P \mid \mathtt{in}(x).P \mid (P \mid Q) \mid !P \\ & \mid \mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q \mid \mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q \end{aligned}$$

where $n \in \mathcal{BN} \cup \mathcal{BK}$, $x \in \mathcal{X}$, and M, N are messages.

Figure 3.2: Syntax for processes.

- ϕ is a substitution with $\mathrm{dom}(\phi) \subseteq \mathcal{AX}$ and for any $x \in \mathrm{dom}(\phi)$, $\phi(x)$ (also denoted $x\phi$) is a message that only contains variables in $\mathrm{dom}(\sigma)$. ϕ represents the terms that have been sent;
- σ is a ground substitution.

The semantics of processes is given through a transition relation $\xrightarrow{\alpha}$, defined in Figure 3.3 (τ denotes a silent action). The relation $\xrightarrow{w}_*$ is defined as the reflexive transitive closure of $\xrightarrow{\alpha}$, where w is the concatenation of all actions. We also write equality up to silent actions $=_\tau$.

Intuitively, process $\mathtt{new}\ n.P$ creates a fresh nonce or key, and behaves like P. Process $\mathtt{out}(M).P$ emits M and behaves like P, provided that the evaluation of M is successful. The corresponding message is stored in the frame ϕ, corresponding to the attacker knowledge. A process may input any message that an attacker can forge (rule IN) from her knowledge ϕ, using a recipe R to compute a new message from ϕ. Note that all names are initially assumed to be secret. Process $P \mid Q$ corresponds to the parallel composition of P and Q. Process $\mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q$ behaves like P in which x is replaced by d if d can be successfully evaluated and behaves like Q otherwise. Process $\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q$ behaves like P if M and N correspond to two equal messages and behaves like Q otherwise. The replicated process $!P$ behaves as an unbounded number of copies of P.

A *trace* of a process P is any possible sequence of transitions in the presence of an attacker that may read, forge, and send messages. Formally, the set of traces $\mathrm{trace}(P)$ is defined as follows.

$$\mathrm{trace}(P) = \{(w, \phi, \sigma) | (\{P\}; \emptyset; \emptyset) \xrightarrow{w}_* (\mathcal{P}; \phi; \sigma)\}$$

Example 1. *Consider the private authentication protocol (PA) presented in Section 3.2. The process $P_b(k_b, \mathtt{pk}(k_a))$ corresponding to responder B answering a request from A has already been defined in Section 3.2.3. The process $P_a(k_a, \mathtt{pk}(k_b))$ corresponding A willing to talk to B is:*

$$P_a(k_a, pk_b) = \mathtt{new}\ N_a.\mathtt{out}(\mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b)).\ \mathtt{in}(z)$$

$$(\{P_1 \mid P_2\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P_1, P_2\} \cup \mathcal{P}; \phi; \sigma) \quad \text{PAR}$$

$$(\{\mathtt{0}\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{P}; \phi; \sigma) \quad \text{ZERO}$$

$$(\{\mathtt{new}\ n.P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \text{NEW}$$

$$(\{\mathtt{new}\ k.P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \text{NEWKEY}$$

$$(\{\mathtt{out}(t).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\mathtt{new}\ ax_n.\mathtt{out}(ax_n)} (\{P\} \cup \mathcal{P}; \phi \cup \{t/ax_n\}; \sigma) \quad \text{OUT}$$
if $t\sigma$ is a ground term, $(t\sigma)\downarrow \neq \bot$, $ax_n \in \mathcal{AX}$ and $n = |\phi| + 1$

$$(\{\mathtt{in}(x).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\mathtt{in}(R)} (\{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(R\phi\sigma)\downarrow /x\}) \quad \text{IN}$$
if R is an attacker term such that $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$, and $(R\phi\sigma)\downarrow \neq \bot$

$$(\{\mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(d\sigma)\downarrow /x\}) \quad \text{LET-IN}$$
if $d\sigma$ is ground and $(d\sigma)\downarrow \neq \bot$

$$(\{\mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \text{LET-ELSE}$$
if $d\sigma$ is ground and $(d\sigma)\downarrow = \bot$, *i.e.* d fails

$$(\{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \text{IF-THEN}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground, $(M\sigma)\downarrow \neq \bot$, $(N\sigma)\downarrow \neq \bot$, and $M\sigma = N\sigma$

$$(\{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \text{IF-ELSE}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground and $(M\sigma)\downarrow = \bot$ or $(N\sigma)\downarrow = \bot$ or $M\sigma \neq N\sigma$

$$(\{!P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P, !P\} \cup \mathcal{P}; \phi; \sigma) \quad \text{REPL}$$

Figure 3.3: Semantics

Altogether, a session between A and B is represented by the process:

$$P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a))$$

where $k_a, k_b \in \mathcal{BK}$, which models that the attacker initially does not know k_a, k_b.

An example of a trace describing an "honest" execution, where the attacker does not interfere with the intended run of the protocol, can be written as (tr, ϕ) where

$$tr =_\tau \mathtt{new}\ x_1.\mathtt{out}(x_1).\mathtt{in}(x_1).\mathtt{new}\ x_2.\mathtt{out}(x_2).\mathtt{in}(x_2)$$

and

$$\phi = \{x_1 \mapsto \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, \mathtt{pk}(k_b)), x_2 \mapsto \mathtt{aenc}(\langle N_a, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a))\}.$$

The trace tr describes A outputting the first message of the protocol, which is stored in $\phi(x_1)$. The attacker then simply forwards $\phi(x_1)$ to B. B then performs several silent actions (decrypting the message, comparing its content to $\mathtt{pk}(k_a)$), and outputs a response, which is stored in $\phi(x_2)$ and forwarded to A by the attacker.

$$
\begin{array}{lcl}
l & ::= & \mathtt{LL} \mid \mathtt{HL} \mid \mathtt{HH} \\
KT & ::= & \mathrm{key}^{l}(T) \mid \mathrm{eqkey}^{l}(T) \mid \mathrm{seskey}^{l,a}(T) \text{ with } a \in \{1, \infty\} \\
T & ::= & l \mid T * T \mid T \vee T \mid [\![\tau_n^{l,a} \,;\, \tau_m^{l',a}]\!] \text{ with } a \in \{1, \infty\} \\
 & \mid & KT \mid \mathrm{pkey}(KT) \mid \mathrm{vkey}(KT) \mid (T)_T \mid \{T\}_T
\end{array}
$$

Figure 3.4: Types for terms

3.3.3 Equivalence

When processes evolve, sent messages are stored in a substitution ϕ while the values of variables are stored in σ. A *frame* is simply a substitution ψ where $\mathrm{dom}(\psi) \subseteq \mathcal{AX}$. It represents the knowledge of an attacker. In what follows, we will typically consider $\phi\sigma$.

Intuitively, two sequences of messages are indistinguishable to an attacker if he cannot perform any test that could distinguish them. This is typically modelled as static equivalence [AF01]. Here, we consider of variant of [AF01] where the attacker is also given the ability to observe when the evaluation of a term fails, as defined for example in [CCP13].

Definition 1 (Static Equivalence)**.** *Two ground frames ϕ and ϕ' are statically equivalent if and only if they have the same domain, and for all attacker terms R, S with variables in $\mathrm{dom}(\phi) = \mathrm{dom}(\phi')$, we have*

$$(R\phi =_{\downarrow} S\phi) \iff (R\phi' =_{\downarrow} S\phi')$$

Then two processes P and Q are in equivalence if no matter how the adversary interacts with P, a similar interaction may happen with Q, with equivalent resulting frames.

Definition 2 (Trace Equivalence)**.** *Let P, Q be two processes. We write $P \sqsubseteq_t Q$ if for all $(s, \phi, \sigma) \in \mathrm{trace}(P)$, there exists $(s', \phi', \sigma') \in \mathrm{trace}(Q)$ such that $s =_\tau s'$ and $\phi\sigma$ and $\phi'\sigma'$ are statically equivalent. We say that P and Q are trace equivalent, and we write $P \approx_t Q$, if $P \sqsubseteq_t Q$ and $Q \sqsubseteq_t P$.*

Note that this definition already includes the attacker's behaviour, since processes may input any message forged by the attacker.

Example 2. *As explained in Section 3.2, anonymity is modelled as an equivalence property. Intuitively, an attacker should not be able to know which agents are executing the protocol. In the case of protocol PA, presented in Example 1, the anonymity property can be modelled by the following equivalence:*

$$P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a)) \approx_t P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_c))$$

3.4 A type system for dynamic keys

Types In our type system we give types to pairs of messages – one from the left process and one from the right one. We store the types of nonces, variables, and keys in a typing environment

$$\frac{}{\mathrm{eqkey}^{l}(T) <: \mathrm{key}^{l}(T)}\ (\textsc{SEqKey}) \qquad \frac{}{\mathrm{seskey}^{l,a}(T) <: \mathrm{eqkey}^{l}(T)}\ (\textsc{SSesKey})$$

$$\frac{}{\mathrm{key}^{l}(T) <: l}\ (\textsc{SKey}) \qquad \frac{T <: \mathrm{eqkey}^{l}(T')}{\mathrm{pkey}(T) <: \mathtt{LL}}\ (\textsc{SPubKey}) \qquad \frac{T <: \mathrm{eqkey}^{l}(T')}{\mathrm{vkey}(T) <: \mathtt{LL}}\ (\textsc{SVKey})$$

$$\frac{T <: T'}{(T)_{T''} <: (T')_{T''}}\ (\textsc{SEnc}) \qquad \frac{T <: T'}{\{T\}_{T''} <: \{T'\}_{T''}}\ (\textsc{SAEnc})$$

Figure 3.5: Selected subtyping rules

Γ. While we store a type for a single nonce or variable occurring in both processes, we assign a potentially different type to every different combination of keys (k, k') used in the left and right process – so called *bikeys*. This is an important non-standard feature that enables us to type protocols using different encryption and decryption keys.

The types for messages are defined in Fig. 3.4 and explained below. Selected subtyping rules are given in Fig. 3.5. We assume three security labels $\mathtt{HH}, \mathtt{HL}$ and $\mathtt{LL}$, ranged over by l, whose first (resp. second) component denotes the confidentiality (resp. integrity) level. Intuitively, values of high confidentiality may never be output to the network in plain, and values of high integrity are guaranteed not to originate from the attacker. Pair types $T * T'$ describe the type of their components and the type $T \vee T'$ is given to messages that can have type T or type T'.

The type $\tau_n^{l,a}$ describes nonces and constants of security level l: the label a ranges over $\{\infty, 1\}$, denoting whether the nonce is bound within a replication or not (constants are always typed with $a = 1$). We assume a different identifier n for each constant and restriction in the process. The type $\tau_n^{l,1}$ is populated by a single name, (i.e., n describes a constant or a non-replicated nonce) and $\tau_n^{l,\infty}$ is a special type, that is instantiated to $\tau_{n_j}^{l,1}$ in the jth replication of the process. Type $[\![\tau_n^{l,a}\ ;\ \tau_m^{l',a}]\!]$ is a refinement type that restricts the set of possible values of a message to values of type $\tau_n^{l,a}$ on the left and type $\tau_m^{l',a}$ on the right. For a refinement type $[\![\tau_n^{l,a}\ ;\ \tau_n^{l,a}]\!]$ with equal types on both sides we write $\tau_n^{l,a}$.

Keys can have three different types ranged over by KT, ordered by a subtyping relation (SEqKey, SSesKey): $\mathrm{seskey}^{l,a}(T) <: \mathrm{eqkey}^{l}(T) <: \mathrm{key}^{l}(T)$. For all three types, l denotes the security label (SKey) of the key and T is the type of the payload that can be encrypted or signed with these keys. This allows us to transfer typing information from one process to another one: e.g. when encrypting, we check that the payload type is respected, so that we can be sure to get a value of the payload type upon decryption. The three different types encode different relations between the left and the right component of a bikey (k, k'). While type $\mathrm{key}^{l}(T)$ can be given to bikeys with different components $k \neq k'$, type $\mathrm{eqkey}^{l}(T)$ ensures that the keys are equal on both sides in the specific typed instruction. Type $\mathrm{seskey}^{l,a}(T)$ additionally guarantees that the key is always the same on the left and the right throughout the whole process. We allow for dynamic generation of keys of type $\mathrm{seskey}^{l,a}(T)$ and use a label a to denote whether the key is generated under replication or not – just like for nonce types.

For a key of type T, we use types $\mathrm{pkey}(T)$ and $\mathrm{vkey}(T)$ for the corresponding public key and verification key, and types $(T')_T$ and $\{T'\}_T$ for symmetric and asymmetric encryptions of messages of type T' with this key. Public keys and verification keys can be treated as LL if the corresponding keys are equal (SPUBKEY,SVKEY) and subtyping on encryptions is directly induced by subtyping of the payload types (SENC, SAENC).

Constraints When typing messages, we generate constraints of the form $(M \sim N)$, meaning that the attacker may see M and N in the left and right process, respectively, and these two messages are thus required to be indistinguishable.

Due to space reasons we only present a few selected rules that are characteristic of the typing of branching protocols. The omitted rules are similar in spirit to the presented ones or are standard rules for equivalence typing [CGLM17a].

3.4.1 Typing messages

The typing judgement for messages is of the form $\Gamma \vdash M \sim N : T \to c$ which reads as follows: under the environment Γ, M and N are of type T and either this is a high confidentiality type (i.e., M and N are not disclosed to the attacker) or M and N are indistinguishable for the attacker assuming the set of constraints c is consistent.

Confidential nonces can be given their label from the typing environment in rule TNONCE. Since their label prevents them from being released in clear, the attacker cannot observe them and we do not need to add constraints for them. They can however be output in encrypted form and will then appear in the constraints of the encryption. Public nonces (labeled as LL) can be typed if they are equal on both sides (rule TNONCEL). These are standard rules, as well as the rules TVAR, TSUB, TPAIR and THIGH [CGLM17a].

A non-standard rule that is crucial for the typing of branching protocols is rule TKEY. As the typing environment contains types for bikeys (k, k') this rule allows us to type two potentially different keys with their type from the environment. With the standard rule TPUBKEYL we can only type a public key of the same keys on both sides, while rule TPUBKEY allows us to type different public keys $\mathtt{pk}(M), \mathtt{pk}(N)$, provided we can show that there exists a valid key type for the terms M and N. This highlights another important technical contribution of this work, as compared to existing type systems for equivalence: we do not only support a fixed set of keys, but also allow for the usage of keys in variables, that have been received from the network.

To show that a message is of type $\{T\}_{T'}$ – a message of type T encrypted asymmetrically with a key of type T', we have to show that the corresponding terms have exactly these types in rule TAENC. The generated constraints are simply propagated. In addition we need to show that T' is a valid type for a public key, or LL, which models untrusted keys received from the network. Note, that this rule allows us to encrypt messages with different keys in the two processes. For encryptions with honest keys (label HH) we can use rule TAENC to give type LL to the messages, if we can show that the payload type is respected. In this case we add the entire encryptions to the constraints, since the attacker can check different encryptions for equality, even if he cannot obtain the plaintext. Rule TAENCL allows us to give type LL to encryptions even if we do not

$$\frac{\Gamma(n) = \tau_n^{l,a} \quad \Gamma(m) = \tau_m^{l,a} \quad l \in \{\mathtt{HH}, \mathtt{HL}\}}{\Gamma \vdash n \sim m : l \to \emptyset}\ (\textsc{TNonce}) \qquad \frac{\Gamma(n) = \tau_n^{\mathtt{LL},a}}{\Gamma \vdash n \sim n : \mathtt{LL} \to \emptyset}\ (\textsc{TNonceL})$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \sim x : T \to \emptyset}\ (\textsc{TVar}) \qquad \frac{\Gamma \vdash M \sim N : T' \to c \qquad T' <: T}{\Gamma \vdash M \sim N : T \to c}\ (\textsc{TSub})$$

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c'}{\Gamma \vdash \langle M, M' \rangle \sim \langle N, N' \rangle : T * T' \to c \cup c'}\ (\textsc{TPair})$$

$$\frac{M, N \text{ well formed}}{\Gamma \vdash M \sim N : \mathtt{HL} \to \emptyset}\ (\textsc{THigh})$$

$$\frac{\Gamma(k, k') = T}{\Gamma \vdash k \sim k' : T \to \emptyset}\ (\textsc{TKey}) \qquad \frac{k \in \mathrm{keys}(\Gamma) \cup \mathcal{FK}}{\Gamma \vdash \mathtt{pk}(k) \sim \mathtt{pk}(k) : \mathtt{LL} \to \emptyset}\ (\textsc{TPubKeyL})$$

$$\frac{\Gamma \vdash M \sim N : T \to \emptyset \qquad \exists T', l. T <: \mathrm{key}^l(T')}{\Gamma \vdash \mathtt{pk}(M) \sim \mathtt{pk}(N) : \mathrm{pkey}(T) \to \emptyset}\ (\textsc{TPubKey})$$

$$\frac{\begin{array}{c}\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c' \\ T' = \mathtt{LL} \ \vee \ (\exists T'', T''', l. T' = \mathrm{pkey}(T'') \wedge T'' <: \mathrm{key}^l(T'''))\end{array}}{\Gamma \vdash \mathtt{aenc}(M, M') \sim \mathtt{aenc}(N, N') : \{T\}_{T'} \to c \cup c'}\ (\textsc{TAenc})$$

$$\frac{\Gamma \vdash M \sim N : \{T\}_{\mathrm{pkey}(T')} \to c \qquad T' <: \mathrm{key}^{\mathtt{HH}}(T)}{\Gamma \vdash M \sim N : \mathtt{LL} \to c \cup \{M \sim N\}}\ (\textsc{TAencH})$$

$$\frac{\Gamma \vdash M \sim N : \{\mathtt{LL}\}_T \to c \qquad (T = \mathrm{pkey}(T') \wedge T' <: \mathrm{eqkey}^l(T'')) \text{ or } T = \mathtt{LL}}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}\ (\textsc{TAencL})$$

Figure 3.6: Selected rules for messages

respect the payload type, or if the key is corrupted. However, we then have to type the plaintexts with type LL since we cannot guarantee their confidentiality. Additionally, we have to ensure that the same key is used in both processes, because the attacker might possess the corresponding private keys and test which decryption succeeds. Since we already add constraints for giving type LL to the plaintext, we do not need to add any additional constraints.

3.4.2 Typing processes

From now on, we assume that processes assign a type to freshly generated nonces and keys. That is, $\mathtt{new}\ n.P$ is now of the form $\mathtt{new}\ n : T.\ P$. This requires a (very light) type annotation from the user. The typing judgement for processes is of the form $\Gamma \vdash P \sim Q \to C$ and can be interpreted as follows: If two processes P and Q can be typed in Γ and if the generated constraint set C is consistent, then P and Q are trace equivalent. We present selected rules in Fig. 3.7.

$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c}{\Gamma \vdash \mathtt{out}(M).P \sim \mathtt{out}(N).Q \to C \cup_\forall c} \text{(POUT)}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \text{ does not contain union types}}{\Gamma \vdash \mathtt{0} \sim \mathtt{0} \to (\emptyset, \Gamma)} \text{(PZERO)} \qquad \frac{\Gamma, x : \mathtt{LL} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{in}(x).P \sim \mathtt{in}(x).Q \to C} \text{(PIN)}$$

$$\frac{\Gamma, n : \tau_n^{l,a} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{new}\ n : \tau_n^{l,a}.P \sim \mathtt{new}\ n : \tau_n^{l,a}.Q \to C} \text{(PNEW)}$$

$$\frac{\Gamma, (k,k) : \mathrm{seskey}^{l,a}(T) \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{new}\ k : \mathrm{seskey}^{l,a}(T).P \sim \mathtt{new}\ k : \mathrm{seskey}^{l,a}(T).Q \to C} \text{(PNEWKEY)}$$

$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash P \mid P' \sim Q \mid Q' \to C \cup_\times C'} \text{(PPAR)}$$

$$\frac{\Gamma \vdash_d t \sim t' : T \qquad \Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = t'\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to C \cup C'} \text{(PLET)}$$

(PLETADECSAME)

$$\frac{\begin{array}{c}\Gamma(y) = \mathtt{LL} \qquad \Gamma(k,k) <: \mathrm{key}^{\mathtt{HH}}(T) \qquad \Gamma, x : T \vdash P \sim Q \to C \\ \Gamma, x : \mathtt{LL} \vdash P \sim Q \to C' \qquad \Gamma \vdash P' \sim Q' \to C'' \\ (\forall T'.\forall k' \neq k.\ \Gamma(k,k') <: \mathrm{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P \sim Q' \to C_{k'}) \\ (\forall T'.\forall k' \neq k.\ \Gamma(k',k) <: \mathrm{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P' \sim Q \to C'_{k'})\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ x = \mathtt{adec}(y,k)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = \mathtt{adec}(y,k)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \\ \qquad \to C \cup C' \cup C'' \cup (\bigcup_{k'} C_{k'}) \cup (\bigcup_{k'} C'_{k'})\end{array}}$$

$$\frac{\begin{array}{cc}\Gamma \vdash P \sim Q \to C_1 & \Gamma \vdash P \sim Q' \to C_2 \\ \Gamma \vdash P' \sim Q \to C_3 & \Gamma \vdash P' \sim Q' \to C_4\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \\ \qquad \to C_1 \cup C_2 \cup C_3 \cup C_4\end{array}} \text{(PIFALL)}$$

Figure 3.7: Selected rules for processes

Rule POUT states that we can output messages to the network if we can type them with type LL, i.e., they are indistinguishable to the attacker, provided that the generated set c of constraints is consistent. The constraints of c are then added to all constraints in the constraint set C. We define $C \cup_\forall c' := \{(c \cup c', \Gamma) \mid (c, \Gamma) \in C\}$. This rule, as well as the rules PZERO, PIN, PNEW, PPAR, and PLET, are standard rules [CGLM17a].

Rule PNEWKEY allows us to generate new session keys at runtime, which models security protocols more faithfully. It also allows us to generate infinitely many keys, by introducing new keys under replication.

Rule PLETADECSAME treats asymmetric decryptions where we use the same fixed honest key

(label HH) for decryptions in both processes. Standard type systems for equivalence have a simplifying (and restrictive) invariant that guarantees that encryptions are always performed using the same keys in both processes and hence guarantee that both processes always take the same branch in decryption (compare rule PLET). In our system however, we allow encryptions with potentially different keys, which requires cross-case validation in order to retain soundness. Still, the number of possible combinations of encryption keys is limited by the assignments in the typing environment Γ. To cover all the possibilities, we type the following combinations of continuation processes:

- Both `then` branches: In this case we know that key k was used for encryption on both sides. Because of $\Gamma(k, k) = \mathrm{key}^{\mathtt{HH}}(T)$, we know that in this case the payload type is T and we type the continuation with $\Gamma, x : T$.
  Because the message may also originate from the attacker (who also has access to the public key), we have to type the two `then` branches also with $\Gamma, x : \mathtt{LL}$.
- Both `else` branches: If decryption fails on both sides, we type the two `else` branches without introducing any new variables.
- Left `then`, right `else`: The encryption may have been created with key k on the left side and another key k' on the right side. Hence, for each $k' \neq k$, such that $\Gamma(k, k')$ maps to a key type with label HH and payload type T', we have to typecheck the left `then` branch and the right `else` branch with $\Gamma, x : T'$.
- Left `else`, right `then`: This case is analogous to the previous one.

The generated set of constraints is simply the union of all generated constraints for the subprocesses. Rule PIFALL lets us typecheck any conditional by simply checking the four possible branch combinations. In contrast to the other rules for conditionals that we present in a companion technical report, this rule does not require any other preconditions or checks on the terms M, M', N, N'.

Destructor Rules The rule PLET requires that a destructor application succeeds or fails equally in the two processes. To ensure this property, it relies on additional rules for destructors. We present selected rules in Fig. 3.8. Rule DADECL is a standard rule that states that a decryption of a variable of type LL with an untrusted key (label LL) yields a result of type LL. Decryption with a trusted (label HH) session key gives us a value of the key's payload type or type LL in case the encryption was created by the attacker using the public key. Here it is important that the key is of type $\mathrm{seskey}^{\mathtt{HH},a}(T)$, since this guarantees that the key is never used in combination with a different key and hence decryption will always equally succeed or fail in both processes. Rule DADECL' is similar to rule DADECL except it uses a variable for decryption instead of a fixed key. Rule DADECT treats the case in which we know that the variable x is an asymmetric encryption of a specific type. If the type of the key used for decryption matches the key type used for encryption, we know the exact type of the result of a successful decryption. DADECT' is similar to DADECT, with a variable as key. In a companion technical report we present similar rules for symmetric decryption and verification of signatures.

$$\frac{\Gamma(k,k) <: \mathrm{key}^{\mathtt{LL}}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash_d \mathtt{adec}(x,k) \sim \mathtt{adec}(x,k) : \mathtt{LL}} \text{ (DADECL)}$$

$$\frac{\Gamma(y) = \mathrm{seskey}^{\mathtt{HH},a}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash_d \mathtt{adec}(x,y) \sim \mathtt{adec}(x,y) : T \vee \mathtt{LL}} \text{ (DADECH')}$$

$$\frac{(\Gamma(y) = \mathrm{seskey}^{\mathtt{LL},a}(T) \vee \Gamma(y) = \mathtt{LL}) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash_d \mathtt{adec}(x,y) \sim \mathtt{adec}(x,y) : \mathtt{LL}} \text{ (DADECL')}$$

$$\frac{\Gamma(k,k) = \mathrm{seskey}^{l,a}(T') \qquad \Gamma(x) = \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T'))}}{\Gamma \vdash_d \mathtt{adec}(x,k) \sim \mathtt{adec}(x,k) : T} \text{ (DADECT)}$$

$$\frac{\Gamma(y) = \mathrm{seskey}^{l,a}(T') \qquad \Gamma(x) = \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T'))}}{\Gamma \vdash_d \mathtt{adec}(x,y) \sim \mathtt{adec}(x,y) : T} \text{ (DADECT')}$$

Figure 3.8: Selected destructor rules

3.4.3 Typing the private authentication protocol

We now show how our type system can be applied to type the Private Authentication protocol presented in section 3.2.3, by showing the most interesting parts of the derivation. We type the protocol using the initial environment Γ presented in Fig. 3.1.

We focus on the responder process P_b and start with the asymmetric decryption. As we use the same key k_b in both processes, we apply rule PLETADECSAME. We have $\Gamma(x) = \mathtt{LL}$ by rule PIN and $\Gamma(k_b, k_b) = \mathrm{key}^{\mathtt{HH}}(\mathtt{HH}, \mathtt{LL})$. We do not have any other entry using key k_b in Γ. We hence typecheck the two `then` branches once with $\Gamma, y : (\mathtt{HH} * \mathtt{LL})$ and once with $\Gamma, y : \mathtt{LL}$, as well as the two `else` branches (which are just `0` in this case).

Typing the let expressions is straightforward using rule PLET. In the conditional we check $y_2 = \mathtt{pk}(k_a)$ in the left process and $y_2 = \mathtt{pk}(k_c)$ in the right process. Since we cannot guarantee which branches are taken or even if the same branch is taken in the two processes, we use rule PIFALL to typecheck all four possible combinations of branches. We now focus on the case where A is successfully authenticated in the left process and is rejected in the right process. We then have to typecheck B's positive answer together with the decoy message: $\Gamma \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_c, \mathtt{pk}(k)) : \mathtt{LL}$.

Fig. 3.9 presents the type derivation for this example. We apply rule TAENC to give type LL to the two terms, adding the two encryptions to the constraint set. Using rule TAENCH we can show that the encryptions are well-typed with type $\{\mathtt{HL}\}_{\mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL}))}$. The type of the payload is trivially shown with rule THIGH. To type the public key, we use rule TPUBKEY followed by rule TKEY, which looks up the type for the bikey (k_a, k) in the typing environment Γ.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{* = \langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, N_b \text{ well formed}}\ \textsc{THigh}
}{\Gamma \vdash \langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle \sim N_b : \mathtt{HL} \to \emptyset}
}{\Gamma(k_a, k) = \mathrm{key}^{\mathtt{HH}}(\mathtt{HL})}\ \textsc{TKey}
}{\Gamma \vdash k_a \sim k : \mathrm{key}^{\mathtt{HH}}(\mathtt{HL}) \to \emptyset}\ \textsc{TPubKey}
}{\Gamma \vdash \mathtt{pk}(k_a) \sim \mathtt{pk}(k) : \mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})) \to \emptyset}\ \textsc{TAenc}
}{\Gamma \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)) : \{\mathtt{HL}\}_{\mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL}))} \to \emptyset}\ \textsc{TAencH}
$$

$$
\Gamma \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)) : \mathtt{LL} \to C
$$

where $C = \{\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\}$.

Figure 3.9: Type derivation for the response to A and the decoy message

3.5 Consistency

Our type system collects constraints that intuitively correspond to (symbolic) messages that the attacker may see (or deduce). Therefore, two processes are in trace equivalence only if the collected constraints are in static equivalence for any plausible instantiation.

However, checking static equivalence of symbolic frames for any instantiation corresponding to a real execution may be as hard as checking trace equivalence [CCD13]. Conversely, checking static equivalence for *any* instantiation may be too strong and may prevent proving equivalence of processes. Instead, we use again the typing information gathered by our type system and we consider only instantiations that comply with the type. Actually, we even restrict our attention to instantiations where variables of type LL are only replaced by deducible terms. This last part is a key ingredient for considering processes with dynamic keys. Hence, we define a constraint to be *consistent* if the corresponding two frames are in static equivalence for any instantiation that can be typed and produces constraints that are included in the original constraint.

Formally, we first introduce the following ingredients:

- $\phi_\ell(c)$ and $\phi_r(c)$ denote the frames that are composed of the left and the right terms of the constraints respectively (in the same order).
- $\phi^\Gamma_{\mathtt{LL}}$ denotes the frame that is composed of all low confidentiality nonces and keys in Γ, as well as all public encryption keys and verification keys in Γ. This intuitively corresponds to the initial knowledge of the attacker.
- Two ground substitutions σ, σ' are well-typed in Γ with constraint c_σ if they preserve the types for variables in Γ, *i.e.*, for all x, $\Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$, and $c_\sigma = \bigcup_{x \in \mathrm{dom}(\Gamma)} c_x$.

The instantiation of a constraint is defined as expected. If c is a set of constraints, and σ, σ' are two substitutions, let $[\![c]\!]_{\sigma,\sigma'}$ be the instantiation of c by σ on the left and σ' on the right, that is, $[\![c]\!]_{\sigma,\sigma'} = \{M\sigma \sim N\sigma' \mid M \sim N \in c\}$.

Definition 3 (Consistency)**.** *A set of constraints c is* consistent *in an environment Γ if for all substitutions σ,σ' well-typed in Γ with a constraint c_σ such that $c_\sigma \subseteq [\![c]\!]_{\sigma,\sigma'}$, the frames*

$\phi^{\Gamma}_{\mathrm{LL}} \cup \phi_{\ell}(c)\sigma$ and $\phi^{\Gamma}_{\mathrm{LL}} \cup \phi_{r}(c)\sigma'$ are statically equivalent. We say that (c, Γ) is consistent if c is consistent in Γ and that a constraint set C is consistent in Γ if each element $(c, \Gamma) \in C$ is consistent.

Compared to [CGLM17a], we now require $c_\sigma \subseteq [\![c]\!]_{\sigma,\sigma'}$. This means that instead of considering any (well typed) instantiations, we only consider instantiations that use fragments of the constraints. For example, this now imposes that low variables are instantiated by terms deducible from the constraint. This refinement of consistency provides a tighter definition and is needed for non fixed keys, as explained in the next section.

3.6 Soundness

In this section, we provide our main results. First, soundness of our type system: whenever two processes can be typed with consistent constraints, then they are in trace equivalence. Then we show how to automatically prove consistency. Finally, we explain how to lift these two first results from finite processes to processes with replication. But first, we discuss why we cannot directly apply the results from [CGLM17a] developed for processes with long term keys.

3.6.1 Example

Consider the following example, typical for a key-exchange protocol: Alice receives some key and uses it to encrypt, e.g. a nonce. Here, we consider a semi-honest session, where an honest agent A is receiving a key from a dishonest agent D. Such sessions are typically considered in combination with honest sessions.

$$\begin{array}{lcll} C & \to & A & : \ \mathtt{aenc}(\langle k, C\rangle, \mathtt{pk}(A)) \\ A & \to & C & : \ \mathtt{aenc}(n, k) \end{array}$$

The process modelling the role of Alice is as follows.

$$\begin{array}{ll} P_A = & \mathtt{in}(x).\ \mathtt{let}\ x' = \mathtt{adec}(x, k_A)\ \mathtt{in}\ \mathtt{let}\ y = \pi_1(x')\ \mathtt{in}\ \mathtt{let}\ z = \pi_2(x')\ \mathtt{in} \\ & \mathtt{if}\ z = C\ \mathtt{then}\ \mathtt{new}\ n.\ \mathtt{out}(\mathtt{enc}(n, y)) \end{array}$$

When type-checking $P_A \sim P_A$ (as part as a more general process with honest sessions), we would collect the constraint $\mathtt{enc}(n, y) \sim \mathtt{enc}(n, y)$ where y comes from the adversary and is therefore a low variable (that is, of type LL). The approach of [CGLM17a] consisted in opening messages as much as possible. In this example, this would yield the constraint $y \sim y$ which typically renders the constraint inconsistent, as exemplified below.

When typechecking the private authentication protocol, we obtain constraints that contain $\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))$ (as seen in Fig. 3.9), where y_1 has type HL. Assume now that the constraint also contains $y \sim y$ for some variable y of type LL and consider the following instantiations of y and y_1: $\sigma(y_1) = \sigma'(y_1) = a$ for some constant a and

$\sigma(y) = \sigma'(y) = \mathtt{aenc}(N_b, \mathtt{pk}(k))$. Note that such an instantiation complies with the type since $\Gamma \vdash \sigma(y) \sim \sigma'(y) : \mathtt{LL} \to c$ for some constraint c. The instantiated constraint would then contain

$$\{\mathtt{aenc}(\langle a, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)),\\ \mathtt{aenc}(N_b, \mathtt{pk}(k)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\}$$

and the corresponding frames are not statically equivalent, which makes the constraint inconsistent for the consistency definition of [CGLM17a] .

Therefore, our first idea consists in proving that we only collect constraints that are saturated w.r.t. deduction: any deducible subterm can already be constructed from the terms of the constraint. Second, we show that for any execution, low variables are instantiated by terms deducible from the constraints. This guarantees that our new notion of consistency is sound. The two results are reflected in the next section.

3.6.2 Soundness

Our type system, together with consistency, implies trace equivalence.

Theorem 1 (Typing implies trace equivalence)**.** *For all P, Q, and C, for all Γ containing only keys, if $\Gamma \vdash P \sim Q \to C$ and C is consistent, then $P \approx_t Q$.*

Example 3. *We can typecheck PA, that is*

$$\Gamma \vdash P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a)) \sim P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_c)) \to C_{PA}$$

where Γ has been defined in Fig. 3.1 and assuming that nonce N_a of process P_a has been annotated with type $\tau_{N_a}^{\mathtt{HH},1}$ and nonce N_b of P_b has been annotated with type $\tau_{N_b}^{\mathtt{HH},1}$. The constraint set C_{PA} can be proved to be consistent using the procedure presented in the next section. Therefore, we can conclude that

$$P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_a)) \approx_t P_a(k_a, \mathtt{pk}(k_b)) \mid P_b(k_b, \mathtt{pk}(k_c))$$

which shows anonymity of the private authentication protocol.

The first key ingredient in the proof of Theorem 1 is the fact that any well-typed low term is deducible from the constraint generated when typing it.

Lemma 1 (Low terms are recipes on their constraints)**.** *For all ground messages M, N, for all Γ, c, if $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ then there exists an attacker recipe R without destructors such that $M = R(\phi_\ell(c) \cup \phi_{\mathtt{LL}}^\Gamma)$ and $N = R(\phi_r(c) \cup \phi_{\mathtt{LL}}^\Gamma)$.*

The second key ingredient is a finer invariant on protocol executions: for any typable pair of processes P, Q, any execution of P can be mimicked by an execution of Q such that low variables are instantiated by well-typed terms constructible from the constraint.

Lemma 2. *For all processes P, Q, for all ϕ, σ, for all multisets of processes $\mathcal{P}$, constraint sets C, sequences s of actions, for all Γ containing only keys, if $\Gamma \vdash P \sim Q \to C$, C is consistent, and $(\{P\}, \emptyset, \emptyset) \xrightarrow{s}_* (\mathcal{P}, \phi, \sigma)$, then there exist a sequence s' of actions, a multiset $\mathcal{Q}$, a frame ϕ', a substitution σ', an environment Γ', a constraint c such that:*
– $(\{Q\}, \emptyset, \emptyset) \xrightarrow{s'}_* (\mathcal{Q}, \phi', \sigma')$, *with* $s =_\tau s'$
– $\Gamma' \vdash \phi\sigma \sim \phi'\sigma' : \mathtt{LL} \to c$, *and for all* $x \in \mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma')$, *there exists* c_x *such that* $\Gamma' \vdash \sigma(x) \sim \sigma(x) : \Gamma'(x) \to c_x$ *and* $c_x \subseteq c$.

Note that this finer invariant guarantees that we can restrict our attention to the instantiations considered for defining consistency.

As a by-product, we obtain a finer type system for equivalence, even for processes with long term keys (as in [CGLM17a]). For example, we can now prove equivalence of processes where some agent signs a low message that comes from the adversary. In such a case, we collect $\mathtt{sign}(x, k) \sim \mathtt{sign}(x, k)$ in the constraint, where x has type LL, which we can now prove to be consistent (depending on how x is used in the rest of the constraint).

3.6.3 Procedure for consistency

We devise a procedure $\mathtt{check\_const}(C)$ for checking consistency of a constraint C, depicted in Figure 3.10. Compared to [CGLM17a], the procedure is actually simplified. Thanks to Lemmas 1 and 2, there is no need to open constraints anymore. The rest is very similar and works as follows:

- First, variables of refined type $[\![\tau_m^{l,1} ; \tau_n^{l',1}]\!]$ are replaced by m on the left-hand-side of the constraint and n on the right-hand-side.
- Second, we check that terms have the same shape (encryption, signature, hash) on the left and on the right and that asymmetric encryption and hashes cannot be reconstructed by the adversary (that is, they contain some fresh nonce).
- The most important step consists in checking that the terms on the left satisfy the same equalities than the ones on the right. Whenever two left terms M and N are unifiable, their corresponding right terms M' and N' should be equal after applying a similar instantiation.

For constraint sets without infinite nonce types, check_const entails consistency.

Theorem 2. *Let C be a set of constraints such that*

$$\forall (c, \Gamma) \in C.\ \forall l, l', m, p.\ \Gamma(x) \neq [\![\tau_m^{l,\infty} ; \tau_p^{l',\infty}]\!].$$

If $\mathtt{check\_const}(C) = \mathtt{true}$, *then C is consistent.*

Example 4. *Continuing Example 3, typechecking the PA protocol yields the set C_{PA} of constraint sets. C_{PA} contains in particular the set*

$$\{\ \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, \mathtt{pk}(k_b)) \sim \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, \mathtt{pk}(k_b)),$$
$$\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\ \}$$

$\texttt{step1}_\Gamma(c) := (\llbracket c \rrbracket_{\sigma_F, \sigma'_F}, \Gamma')$, with $F := \{x \in \mathrm{dom}(\Gamma) \mid \exists m, n, l, l'.\ \Gamma(x) = \llbracket \tau_m^{l,1}\ ;\ \tau_n^{l',1} \rrbracket\}$ and σ_F, σ'_F defined by

$$\begin{cases} \bullet\ \mathrm{dom}(\sigma_F) = \mathrm{dom}(\sigma'_F) = F \\ \bullet\ \forall x \in F.\ \forall m, n, l, l'. \Gamma(x) = \llbracket \tau_m^{l,1}\ ;\ \tau_n^{l',1} \rrbracket \Rightarrow \sigma_F(x) = m\ \wedge\ \sigma'_F(x) = n \end{cases}$$

and Γ' is $\Gamma|_{\mathrm{dom}(\Gamma)\setminus F}$ extended with $\Gamma'(n) = \tau_n^{l,1}$ for all nonce n such that $\tau_n^{l,1}$ occurs in Γ.

$\texttt{step2}_\Gamma(c) :=$ check that for all $M \sim N \in c$, M and N are both

- $\texttt{enc}(M', M'')$, $\texttt{enc}(N', N'')$ where M'', N'' are either
  - keys k, k' where $\exists T. \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - or a variable x such that $\exists T. \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
- or encryptions $\texttt{aenc}(M', M'')$, $\texttt{aenc}(N', N'')$ where
  - M' and N' contain directly under pairs a nonce n such that $\Gamma(n) = \tau_n^{\mathtt{HH},a}$ or a secret key k such that $\exists T, k'. \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$ or $\Gamma(k', k) <: \mathrm{key}^{\mathtt{HH}}(T)$, or a variable x such that $\exists m, n, a. \Gamma(x) = \llbracket \tau_m^{\mathtt{HH},a}\ ;\ \tau_n^{\mathtt{HH},a} \rrbracket$, or a variable x such that $\exists T. \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - M'' and N'' are either
    - public keys $\texttt{pk}(k)$, $\texttt{pk}(k')$ where $\exists T. \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
    - or public keys $\texttt{pk}(x)$, $\texttt{pk}(x)$ where $\exists T. \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
    - or a variable x such that $\exists T, T'. \Gamma(x) = \mathrm{pkey}(T)$ and $T <: \mathrm{key}^{\mathtt{HH}}(T')$;
- or hashes $\texttt{h}(M')$, $\texttt{h}(N')$, where M', N' similarly contain a secret value under pairs;
- or signatures $\texttt{sign}(M', M'')$, $\texttt{sign}(N', M'')$ where M'', N'' are either
  - keys k, k' where $\exists T. \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - or a variable x such that $\exists T. \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;

$\texttt{step3}_\Gamma(c) :=$ If for all $M \sim M'$ and $N \sim N' \in c$ such that M, N are unifiable with a most general unifier μ, and such that $\forall x \in \mathrm{dom}(\mu). \exists l, l', m, p.\ (\Gamma(x) = \llbracket \tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty} \rrbracket) \Rightarrow (x\mu \in \mathcal{X}\ \vee\ \exists i.\ x\mu = m_i)$ we have $M'\alpha\theta = N'\alpha\theta$ where $\forall x \in \mathrm{dom}(\mu). \forall l, l', m, p, i. (\Gamma(x) = \llbracket \tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty} \rrbracket\ \wedge\ \mu(x) = m_i) \Rightarrow \theta(x) = p_i$
and α is the restriction of μ to $\{x \in \mathrm{dom}(\mu) \mid \Gamma(x) = \mathtt{LL}\ \wedge\ \mu(x) \in \mathcal{N}\}$;
and if the symmetric condition for the case where M', N' are unifiable holds, too, return $\mathtt{true}$.

$\texttt{check\_const}(C) :=$ for all $(c, \Gamma) \in C$, let $(c_1, \Gamma_1) := \texttt{step1}_\Gamma(c)$ and check that $\texttt{step2}_{\Gamma_1}(c_1) = \mathtt{true}$ and $\texttt{step3}_{\Gamma_1}(c_1) = \mathtt{true}$.

Figure 3.10: Procedure for checking consistency.

where variable y_1 has type `HL` *(we also have the same constraint but where y_1 has type* `LL`*). The other constraint sets of C_{PA} are similar and correspond to the various cases (*`else` *branch of P_a with* `then` *branch of P_b, etc.). The procedure* `check_const` *returns true since no two terms can be unified, which proves consistency. Similarly, the other constraints generated for PA can be proved to be consistent applying* `check_const`.

3.6.4 From finite to replicated processes

The previous results apply to processes without replication only. In the spirit of [CGLM17a], we lift our results to replicated processes. We proceed in two steps.

1. Whenever $\Gamma \vdash P \sim Q \to C$, we show that: $[\,\Gamma\,]_1 \cup \cdots \cup [\,\Gamma\,]_n \vdash [\,P\,]_1 | \ldots | [\,P\,]_n \sim [\,Q\,]_1 | \ldots | [\,Q\,]_n \to [\,C\,]_1 \cup_\times \cdots \cup_\times [\,C\,]_n$, where $[\,\Gamma\,]_i$ is intuitively a copy of Γ, where variables x have been replaced by x_i, and nonces or keys n of infinite type $\tau_n^{l,\infty}$ (or $\mathrm{seskey}^{l,\infty}(T)$) have been replaced by n_i. The copies $[\,P\,]_i$, $[\,Q\,]_i$, and $[\,C\,]_i$ are defined similarly.

2. We cannot directly check consistency of infinitely many constraints that are of the form $[\,C\,]_1 \cup_\times \cdots \cup_\times [\,C\,]_n$. Instead, we show that it is sufficient to check consistency of two copies $[\,C\,]_1 \cup_\times [\,C\,]_2$ only. The reason why we need two copies (and not just one) is to detect when messages from different sessions may become equal.

Formally, we can prove trace equivalence of replicated processes.

Theorem 3. *Consider P, Q, P' ,Q', C, C', such that P, Q and P', Q' do not share any variable. Consider Γ, containing only keys and nonces with finite types.*

Assume that P and Q only bind nonces and keys with infinite nonce types, i.e. *using* `new` $m : \tau_m^{l,\infty}$ *and* `new` $k : \mathrm{seskey}^{l,\infty}(T)$ *for some label l and type T; while P' and Q' only bind nonces and keys with finite types,* i.e. *using* `new` $m : \tau_m^{l,1}$ *and* `new` $k : \mathrm{seskey}^{l,1}(T)$.

Let us abbreviate by `new` $\overline{n}$ *the sequence of declarations of each nonce $m \in \mathrm{dom}(\Gamma)$ and session key k such that $\Gamma(k,k) = \mathrm{seskey}^{l,1}(T)$ for some l, T. If*

- $\Gamma \vdash P \sim Q \to C$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- $\texttt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \texttt{true}$,

then $\qquad \texttt{new}\ \overline{n}.\ ((!P) \mid P') \approx_t \texttt{new}\ \overline{n}.\ ((!Q) \mid Q')$.

Interestingly, Theorem 3 allows to consider a mix of finite and replicated processes.

3.7 Experimental results

We implemented our typechecker as well as our procedure for consistency in a prototype tool TypeEq. We adapted the original prototype of [CGLM17a] to implement additional cases corresponding to the new typing rules. This also required to design new heuristics w.r.t. the order in which typing rules should be applied. Of course, we also had to support for the new bikey types, and for arbitrary terms as keys. This represented a change of about 40% of the code of the software. We ran our experiments on a single Intel Xeon E5-2687Wv3 3.10GHz core, with 378GB of RAM (shared with the 19 other cores). Actually, our own prototype does not require a large amount of RAM. However, some of the other tools we consider use more than 64GB of RAM on some examples (at which point we stopped the experiment). More precise figures about our tool are provided in the table of Figure 3.11. The corresponding files can be found at [CGLM17c].

We tested TypeEq on two symmetric key protocols that include a handshake on the key (Yahalom-Lowe and Needham-Schroeder symmetric key protocols). In both cases, we prove key usability of the exchanged key. Intuitively, we show that an attacker cannot distinguish between two encryptions of public constants: $P.\mathtt{out}(\mathtt{enc}(a,k)) \approx_t P.\mathtt{out}(\mathtt{enc}(b,k))$. We also consider one standard asymmetric key protocol (Needham-Schroeder-Lowe protocol), showing strong secrecy of the exchanged nonce.

Helios [Adi08] is a well known voting protocol. We show ballot privacy, in the presence of a dishonest board, assuming that voters do not revote (otherwise the protocol is subject to a copy attack [Roe16], a variant of [CS13]). We consider a more precise model than the previous Helios models which assume that voters initially know the election public key. Here, we model the fact that voters actually receive the (signed) freshly generated election public key from the network. The BAC protocol is one of the protocols embedded in the biometric passport [008]. We show anonymity of the passport holder $P(A) \approx_t P(B)$. Actually, the only data that distinguish $P(A)$ from $P(B)$ are the private keys. Therefore we consider an additional step where the passport sends the identity of the agent to the reader, encrypted with the exchanged key. Finally, we consider the private authentication protocol, as described in this paper.

3.7.1 Bounded number of sessions

We first compare TypeEq with the tools for a bounded number of sessions. Namely, we consider Akiss [CCK12], APTE [Che14] as well as its optimised variant with partial order reduction APTE-POR [BDH15], SPEC [DT10], and SatEquiv [CDD17]. We step by step increase the number of sessions until we reach a "complete" scenario where each role is instantiated by A talking to B, A talking to C, B talking to A, and B talking to C, where A, B are honest while C is dishonest. This yields 14 sessions for symmetric-key protocols with two agents and one server, and 8 sessions for a protocol with two agents. In some cases, we further increase the number of sessions (replicating identical scenarios) to better compare tools performance. The results of our experiments are reported in Fig. 3.11. Note that SatEquiv fails to cover several cases because it does not handle asymmetric encryption nor else branches.

Protocols (# sessions)		Akiss	APTE	APTE-POR	Spec	Sat-Eq	TypeEq	
							Time	Memory
Needham - Schroeder (symmetric)	3	4.2s	0.39s	0.086s	59.3s	0.14s	0.006s	4.0 MB
	6	TO	TO	9m22s	TO	0.53s	0.009s	4.7 MB
	10			SO		3.7s	0.012s	5.0 MB
	14					18s	0.015s	6.9 MB
Yahalom - Lowe	3	1.0s	2.9s	0.095s	10s	0.063s	0.006s	3.8 MB
	6	MO	TO	11m20s	MO	0.26s	0.017s	4.9 MB
	10			SO		3.0s	0.015s	4.9 MB
	14					18s	0.019s	5.0 MB
Needham-Schroeder-Lowe	2	0.10s	3.8s	0.06s	28s	x	0.004s	3.1 MB
	4	1m8s	BUG	BUG	TO		0.004s	3.4 MB
	8	TO					0.007s	4.7 MB
Private Authentication	2	0.19s	1.2s	0.034s	x	x	0.004s	3.2 MB
	4	99m	TO	24.6s			0.013s	4.9 MB
	8	MO		TO			1s	37 MB
Helios	3	MO	BUG	BUG	x	x	0.005s	3.5 MB
BAC	2	4.0s	0.20s	0.032s	x	x	0.004s	2.9 MB
	3	SO	185m	2.6s			0.004s	3.1 MB
	5		TO	107m			0.005s	3.4 MB
	7			TO			0.005s	3.8 MB

TO: Time Out (>12h) MO: Memory Overflow (>64GB) SO: Stack Overflow

Figure 3.11: Experimental results for the bounded case

3.7.2 Unbounded number of sessions

We then compare TypeEq with Proverif. As shown in Fig. 3.12, the performances are similar except that ProVerif cannot prove Helios. The reason lies in the fact that Helios is actually subject to a copy attack if voters revote and ProVerif cannot properly handle processes that are executed only once. Similarly, Tamarin cannot properly handle the else branch of Helios (which models that the ballot box rejects duplicated ballots). Tamarin fails to prove that the underlying check either succeeds or fails on both sides.

3.8 Conclusion and discussion

We devise a new type system to reason about keys in the context of equivalence properties. Our new type system significantly enhances the preliminary work of [CGLM17a], covering a larger class of protocols that includes key-exchange protocols, protocols with setup phases, as well as protocols that branch differently depending on the decryption key.

Our type system requires a light type annotation that can be directly inferred from the structure of the messages. As future work, we plan to develop an automatic type inference system. In our

Protocols	ProVerif	TypeEq
Helios	x	0.005s
Needham-Schroeder (sym)	0.23s	0.016s
Needham-Schroeder-Lowe	0.08s	0.008s
Yahalom-Lowe	0.48s	0.020s
Private Authentication	0.034s	0.008s
BAC	0.038s	0.005s

Figure 3.12: Experimental results for an unbounded number of sessions

case study, the only intricate case is the Helios protocol where the user has to write a refined type that corresponds to an over-approximation of any encrypted message. We plan to explore whether such types could be inferred automatically.

We also plan to study how to add phases to our framework, in order to cover more properties (such as unlinkability). This would require to generalize our type system to account for the fact that the type of a key may depend on the phase in which it is used.

Another limitation of our type system is that it does not address processes with too dissimilar structure. While our type system goes beyond diff-equivalence, e.g. allowing else branches to be matched with then branches, we cannot prove equivalence of processes where traces of P are dynamically mapped to traces of Q, depending on the attacker's behaviour. Such cases occur for example when proving unlinkability of the biometric passport. We plan to explore how to enrich our type system with additional rules that could cover such cases, taking advantage of the modularity of the type system.

Conversely, the fact that our type system discards processes that are in equivalence shows that our type system proves something stronger than trace equivalence. Indeed, processes P and Q have to follow some form of uniformity. We could exploit this to prove stronger properties like oblivious execution, probably further restricting our typing rules, in order to prove e.g. the absence of side-channels of a certain form.

CHAPTER 4

Runtime Monitoring for Client Side Web Session Security

Abstract

Micro-policies, originally proposed to implement hardware-level security monitors, constitute a flexible and general enforcement technique, based on assigning security tags to system components and taking security actions based on dynamic checks over these tags. In this paper, we present the first application of micro-policies to web security, by proposing a core browser model supporting them and studying its effectiveness at securing web sessions. In our view, web session security requirements are expressed in terms of a simple, declarative information flow policy, which is then automatically translated into a micro-policy enforcing it. This leads to a browser-side enforcement mechanism which is elegant, sound and flexible, while being accessible to web developers. We show how a large class of attacks against web sessions can be uniformly and effectively prevented by the adoption of this approach. We also develop a proof-of-concept implementation of a significant core of our proposal as a Google Chrome extension, Michrome: our experiments show that Michrome can be easily configured to enforce strong security policies without breaking the functionality of websites.

The work presented in this chapter is a collaboration with Stefano Calzavara, Riccardo Focardi and Matteo Maffei and has been published at published at the 29th IEEE Computer Security Foundations Symposium (CSF'16) under the title "Micro-Policies for Web Session Security" [CFGM16a]. I and Stefano Calzavara contributed equally to the theoretic results presented in the paper and I am responsible for the formal proof presented in Appendix A.2. The experimental evaluation was performed by Stefano Calzavara and Riccardo Focardi.

4.1 Introduction

The Web is nowadays the primary means of access to a plethora of online services with strict security requirements. Electronic health records and online statements of income are a well-established reality as of now, and more and more security-sensitive services are going to be supplied online in the next few years. Despite the critical importance of securing these online services, web applications and, more specifically, *web sessions* are notoriously hard to protect, since they can be attacked at many different layers.

At the network layer, man-in-the-middle attacks can break both the confidentiality and the integrity of web sessions running (at least partially) over HTTP. The standard solution against these attacks is deploying the entire web application over HTTPS with trusted certificates and, possibly, making use of HSTS [HJB12] to prevent subtle attacks like SSL stripping. At the session implementation layer, code injection attacks (or again network attacks) can be exploited to steal authentication cookies and hijack a web session, or to compromise the integrity of the cookie jar and mount dangerous attacks like session fixation [JBSP11]. This is particularly problematic because, though the standard HttpOnly and Secure cookie attributes [Bar11] are effective at protecting cookie confidentiality, no effective countermeasure exists as of now to ensure cookie integrity on the Web [ZJL+15]. Finally, web sessions can also be attacked at the application layer: for instance, since browsers automatically attach cookies set by a website to all the requests sent to it, cross-site request forgery (CSRF) attacks can be mounted by a malicious web page to harm the integrity of the user session with a trusted web application and inject attacker-controlled messages into it. Standard solutions against this problem include the usage of secret tokens and the validation of the Origin header attached by the browser to filter out malicious web requests [BJM08a].

In principle, it is possible to achieve a reasonable degree of security for web sessions using the current technologies, but the overall picture still exhibits several important shortcomings and it is far from being satisfactory. First, there are mechanisms like the HttpOnly cookie attribute which are easy to use, popular and effective, but lack flexibility: a cookie may either be HttpOnly or not, hence JavaScript may either be able to access it or be prevented from doing any kind of computation over the cookie value. There is no way, for instance, to let JavaScript access a cookie for legitimate computations, at the cost of disciplining its communication behaviour to prevent the cookie leakage. Then, there are defenses which are sub-optimal and not always easy to implement: this is the case for token-based protection against CSRF. Not only this approach must be directly implemented into the APIs of a web development framework to ensure that it is convenient to use, but also it is not very robust, since it fails in presence of code injection vulnerabilities which disclose the token value to the attacker. Finally, we observe that some attacks and attack vectors against web sessions are underestimated by existing standards and no effective solution against them can be deployed as of now: this is the case for many threats to cookie integrity [ZJL+15]. These issues will likely be rectified with ad-hoc solutions in future standards, whenever browser vendors and web application developers become more concerned about their importance, and find a proper way to patch them while preserving the compatibility with existing websites.

In this paper, we advocate that a large class of attacks harming the security of web sessions can

be provably, uniformly, and effectively prevented by the adoption of *browser-enforced security policies*, reminiscent of a dynamic typing discipline for the browser. In particular, we argue for the adoption of *micro-policies* [dADG+15] as a convenient tool to improve the security of web sessions, by disciplining the browser behaviour when interacting with security-sensitive web applications. Roughly, the specification of a micro-policy involves: (1) the definition of a set of *tags*, used to label selected elements of the web ecosystem, like URLs, cookies, network connections, etc., and (2) the definition of a *transfer* function, defining which operations are permitted by the browser based on the tags and how tags are assigned to browser elements after a successful operation. This kind of security policies has already proved helpful for deploying hardware-level security monitors and nicely fits existing web security solutions, like cookie security attributes [Bar11] and whitelist-based defenses in the spirit of the Content Security Policy [WBV15].

Though previous work has already proposed browser-side security policies as a viable approach for protecting the Web [JSH07, LV09, SSM10, WBS11, CMKW13], we are the first to carry out a foundational study on a possible extension of a web browser with support for micro-policies and discuss web session security as an important application for this framework. There are many different ways to deploy micro-policies in web browsers, but our proposal is driven by two main design goals aimed at simplifying a large-scale adoption. First, it is *light-weight* and intended to minimize changes to existing web browsers, since it embraces a coarse-grained enforcement approach. Second, it is *practical*: though our proposal is based on a non-trivial theory, we strive for supporting declarative policies for web session security, reminiscent of the tools and the abstractions which web developers already appreciate and use today. We thus propose to express web session security requirements in terms of a simple, declarative information flow policy, which is automatically translated into a micro-policy enforcing it.

To assess the effectiveness of our approach, we developed a proof-of-concept implementation of a significant core of our proposal as a Google Chrome extension, Michrome, and we performed a preliminary experimental evaluation on existing websites. Our experiments show that Michrome can be easily configured to enforce strong security policies without breaking the functionality of websites. We see Michrome as a first reasonable attempt at evaluating the practicality of our theory rather than as a full-fledged defensive mechanism ready for inclusion in standard web browsers. More work is needed to support all the features of our formal model, though we were able to implement and test a significant part of it.

4.1.1 Contributions

Our contributions can be summarized as follows:

1. we design FF^{τ}, a core model of a web browser extended with support for micro-policies. We define the operational behaviour of FF^{τ} using a small-step reactive semantics in the spirit of previous formal work on browser security [BPS+09, BP10, BCF+14]. The semantics of FF^{τ} is parametric with respect to an arbitrary set of tags and the definition of a transfer function operating on these tags;

2. we instantiate the set of tags of FF^τ to intuitive information flow labels and we characterize standard attackers from the web security literature in terms of these labels. We then discuss how to translate simple information flow policies for web session security into micro-policies which enforce them: this is crucial to ensure that most web developers can benefit from our proposal;

3. we discuss example applications of our theory by revisiting known attacks against web sessions and discussing limitations of existing solutions. We then show how these issues are naturally and more effectively solved by our enforcement technique;

4. we develop a prototype implementation of our proposal as a standard Google Chrome extension, Michrome, and we run a set of experiments testing its practicality.

Michrome is available online [CFGM16b].

4.2 Key Ideas

In this section, we give an intuitive overview of the most salient aspects of our framework. We model the browser as a *reactive system*, transforming a stream of input events into a stream of output events. Output events are network requests that are sent by the browser, while input events represent incoming network responses or user actions processable by the browser, e.g., the insertion of a URL into the address bar. Our sets of events include key elements of standard web browsing, like HTTP(S) requests, responses and redirects. For example, the input stream:

$$I = [\mathsf{load}(u), \mathsf{doc\_resp}_n(u : \{\mathsf{ck}(k, v)^\ell\}, \mathsf{unit})],$$

instructs the browser to establish a new network connection n to the URL u and retrieve from that connection a response including a cookie $\mathsf{ck}(k, v)^\ell$ and an empty document unit. The cookie, formally seen as a mapping between key k and value v, has a security *label* ℓ, consisting of a confidentiality policy and an integrity policy. For instance, the confidentiality policy $\{\mathsf{https}(d)\}$ expresses that the value of the cookie should only have a visible import for an attacker who is able to decrypt the HTTPS communication with the domain d setting the cookie.

We argue for the adoption of browser-side micro-policies enforcing this kind of security policies. Security is formalized in terms of *reactive non-interference*, a property dictating that similar input streams must always be transformed into similar output streams. Confidentiality is characterized by identifying suitable similarity relations on input streams, based on what the attacker is able to observe about the corresponding output streams. For instance, consider a network attacker with full control of the HTTP traffic: to formalize that cookies with the confidentiality policy $\{\mathsf{https}(d)\}$ have no visible import for the attacker, the stream similarity on inputs may relate streams which are identical except for the value of these cookies.

As an example, let u_s be a HTTPS URL on domain d, it is safe to consider the following two input streams, differing in the cookie value, as similar:

$$I_1 = [\mathsf{load}(u), \mathsf{doc\_resp}_n(u : \{\mathsf{ck}(k, v)^\ell\}, \mathsf{unit}), \mathsf{load}(u_s)]$$
$$I_2 = [\mathsf{load}(u), \mathsf{doc\_resp}_n(u : \{\mathsf{ck}(k, v')^\ell\}, \mathsf{unit}), \mathsf{load}(u_s)]$$

The reason is that the browser will react to these input streams by producing the following output streams:

$$O_1 = [\mathsf{doc\_req}(u : \emptyset), \bullet, \mathsf{doc\_req}(u_s : \mathsf{ck}(k, v)^\ell)]$$
$$O_2 = [\mathsf{doc\_req}(u : \emptyset), \bullet, \mathsf{doc\_req}(u_s : \mathsf{ck}(k, v')^\ell)]$$

These streams include a document request to u without any cookie, a dummy event ($\bullet$) as a reaction to the empty document, and a document request to u_s including the previously received cookie, which is the normal behaviour of a web browser. Since u_s is a HTTPS URL, the last events of O_1 and O_2 cannot be distinguished by a network attacker, hence the two output streams are similar and there is no violation to reactive non-interference.

But what if the $\mathsf{load}(u_s)$ event in I_1, I_2 was replaced by $\mathsf{load}(u_h)$, where u_h is a HTTP URL on domain d? The behaviour of the browser will be restricted by the underlying micro-policy for non-interference, forcing the production of two output streams not including any cookie in the last event to ensure similarity upon output. These restrictions are enforced by assigning labels to browser components (cookies, connections, scripts...) and by performing runtime label checks upon event processing, reminiscent of a dynamic typing discipline for the browser. Interestingly, simple and intuitive policies like the one we discussed are expressive enough to prevent a large class of known attacks against web sessions. Moreover, despite their simplicity, these policies are actually stronger than currently deployed web solutions (cf. Section 4.6), providing an expressive mechanism to formally define and enforce confidentiality and integrity properties for web sessions.

4.3 Background on Reactive Systems

Web browsers can be formalized using labelled transition systems known as *reactive systems* [BPS+09, BP10]. A reactive system is a state machine which waits for an input, produces outputs in response to it, and repeats the process indefinitely.

Definition 1 (Reactive System)**.** *A* reactive system *is a tuple* $R = \langle \mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, C_0, \longrightarrow \rangle$*, where* $\mathcal{C}$ *and* $\mathcal{P}$ *are disjoint sets of consumer and producer states respectively, while* $\mathcal{I}$ *and* $\mathcal{O}$ *are disjoint sets of input and output events respectively. The consumer state* C_0 *is the initial state of the system and the last component,* $\longrightarrow$*, is a labelled transition relation over the set of states* $\mathcal{Q} \triangleq \mathcal{C} \cup \mathcal{P}$ *and the set of events* $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{O}$*, subject to the following constraints:*

1. *if* $C \in \mathcal{C}$ *and* $C \xrightarrow{a} Q$*, then* $a \in \mathcal{I}$ *and* $Q \in \mathcal{P}$*;*
2. *if* $P \in \mathcal{P}$ *and* $P \xrightarrow{a} Q$ *for some* $Q \in \mathcal{Q}$*, then* $a \in \mathcal{O}$*;*
3. *if* $C \in \mathcal{C}$ *and* $i \in \mathcal{I}$*, then there exists* $P \in \mathcal{P}$ *s.t.* $C \xrightarrow{i} P$*;*
4. *if* $P \in \mathcal{P}$*, then there exist* $o \in \mathcal{O}$ *and* $Q \in \mathcal{Q}$ *s.t.* $P \xrightarrow{o} Q$*.*

We define *streams* of events through the coinductive interpretation of the following grammar: $S ::= [\,] \mid a :: S$. The semantics of a reactive system R is defined in terms of *traces* (I, O), where

I is a stream of input events and O is a stream of output events generated by R as the result of processing I.

Definition 2 (Trace)**.** *Let $R = \langle \mathcal{C}, \mathcal{P}, \mathcal{I}, \mathcal{O}, C_0, \longrightarrow \rangle$ be a reactive system. Given an input stream I, the state $Q \in \mathcal{C} \cup \mathcal{P}$ generates the output stream O iff the judgement $Q(I) \Downarrow O$ can be coinductively derived by the following inference rules:*

$$\text{(C-Nil)}\ \frac{}{C([\,]) \Downarrow [\,]} \qquad \text{(C-In)}\ \frac{C \xrightarrow{i} P \quad P(I) \Downarrow O}{C(i :: I) \Downarrow O} \qquad \text{(C-Out)}\ \frac{P \xrightarrow{o} Q \quad Q(I) \Downarrow O}{P(I) \Downarrow o :: O}$$

We say that R generates the trace *(I, O) iff $C_0(I) \Downarrow O$.*

A natural definition of information security for reactive computations can be formulated using *reactive non-interference* [BPS$^+$09]. We presuppose the existence of a label pre-order $(\mathcal{L}, \sqsubseteq)$ and we represent the attacker as a label $\ell \in \mathcal{L}$, defining its abilities to observe and corrupt data. These abilities are formalized by a label-indexed family of predicates rel_ℓ, identifying security relevant events, and a label-indexed family of similarity relations $\sim_\ell$, identifying indistinguishable events. We collect these two families of relations in a *policy* $\pi = \langle rel_\ell, \sim_\ell \rangle$.

Given a policy π, we define a notion of *similarity* between two streams of events for an attacker ℓ. There are several sensible definitions of similarity in the literature, the one we use here (called *ID-similarity*) leads to a termination-insensitive notion of non-interference and comes with a convenient proof technique based on unwinding relations [BPS$^+$09].

Definition 3 (ID-similarity)**.** *Two streams of events S and S' are* ID-similar *(*similar *for short) for ℓ under $\pi = \langle rel_\ell, \sim_\ell \rangle$ iff the judgement $S \approx^\pi_\ell S'$ can be coinductively derived by the following inference rules:*

$$\text{(S-Empty)}\ [\,] \approx^\pi_\ell [\,] \qquad \text{(S-Match)}\ \frac{rel_\ell(s) \qquad rel_\ell(s') \qquad s \sim_\ell s' \qquad S \approx^\pi_\ell S'}{s :: S \approx^\pi_\ell s' :: S'}$$

$$\text{(S-Left)}\ \frac{\neg rel_\ell(s) \qquad S \approx^\pi_\ell S'}{s :: S \approx^\pi_\ell S'} \qquad \text{(S-Right)}\ \frac{\neg rel_\ell(s) \qquad S \approx^\pi_\ell S'}{S \approx^\pi_\ell s :: S'}$$

Intuitively, a reactive system satisfies non-interference under a policy π if and only if, whenever it is fed two similar input streams, it produces two similar output streams for all the possible attackers (labels).

Definition 4 (Reactive Non-interference)**.** *A reactive system is* non-interferent *under π iff, for all labels ℓ and all its traces (I, O) and (I', O') such that $I \approx^\pi_\ell I'$, we have $O \approx^\pi_\ell O'$.*

Reactive non-interference has been proposed in the past as a useful security baseline to prove protection against common attacks against web sessions, including the theft of authentication cookies [GDNP12, BCFK14, BCFK15] and cross-site request forgery [KCB$^+$14].

4.4 Micro-Policies for Browser-Side Security

Our model FF^τ is inspired by existing formal models for web browsers based on reactive systems [BP10, BCF+14]. It is an extension of the Flyweight Firefox model [BCF+14] with tags and support for enforcing micro-policies based on them.

4.4.1 Syntax

A *map* M is a partial function from keys to values. We let $\{\}$ stand for the empty map and we let $\mathit{dom}(M)$ denote the *domain* of M, i.e., the set of keys bound to a value in M. We let $M_1 \uplus M_2$ be the union of two maps with disjoint domains.

Tags

We presuppose the existence of a denumerable set of tags *Tags* and we let τ range over them. We do not put any restriction on the format of these tags, though we instantiate them to a specific format in the next section.

Terms

We presuppose a set of domain names $\mathcal{D}$ (ranged over by d) and a set of strings $\mathcal{S}$ (ranged over by s). The *signature* for the set of terms $\mathcal{T}$ is:

$$\Sigma = \{\mathsf{http}, \mathsf{https}, \mathsf{url}(\cdot,\cdot,\cdot), \mathsf{ck}(\cdot,\cdot,\cdot)\} \cup \mathcal{D} \cup \mathcal{S} \cup \mathit{Tags}.$$

Let $\mathcal{X}$ be a set of variables and $\mathcal{N}$ be a set of names, the set of terms $\mathcal{T}$ (ranged over by t) is defined as follows: if $t \in \mathcal{X} \cup \mathcal{N}$, then $t \in \mathcal{T}$; if f is an n-ary function symbol in Σ and $\{t_1, \ldots, t_n\} \subseteq \mathcal{T}$, then $f(t_1, \ldots, t_n) \in \mathcal{T}$.

URLs

We let $\mathcal{U} \subseteq \mathcal{T}$ be the set of the URLs, i.e., the set of terms of the form $\mathsf{url}(t, d, s)$ with $t \in \{\mathsf{http}, \mathsf{https}\}$. Given $u = \mathsf{url}(t, d, s)$, let $\mathit{prot}(u) = t$, $\mathit{host}(u) = d$ and $\mathit{path}(u) = s$. We assume that each URL $u \in \mathcal{U}$ comes with an associated tag, returned by a function $\mathit{tag} : \mathcal{U} \to \mathit{Tags}$. For instance, the *tag* function may assign the Secure tag to HTTPS pages and the Insecure tag to HTTP pages: this information can be used to apply different micro-policies in the browser.

Cookies

We let $\mathcal{CK} \subseteq \mathcal{T}$ be the set of cookies, i.e., the set of terms of the form $\mathsf{ck}(s, s', \tau)$. Formally, cookies are just key-value pairs (s, s') extended with a tag τ. We assume this tag is assigned by a function $\kappa : \mathcal{D} \times \mathcal{S} \to \mathit{Tags}$, so that cookies with the same key set by the same domain must have the same tag. We typically use the more evocative notation $\mathsf{ck}(k, v)^\tau$ to represent cookies. Given $ck = \mathsf{ck}(k, v)^\tau$, we let $\mathit{key}(ck) = k$ and $\mathit{value}(ck) = v$.

Scripts

We let values v, expressions e and scripts scr be defined by the following productions:

$$\begin{array}{llll} \textit{Values} & v & ::= & t \mid \mathsf{unit} \mid \lambda x.e \\ \textit{Expr.} & e & ::= & v\, v' \mid \mathsf{let}\ x = e\ \mathsf{in}\ e' \mid \mathsf{get\text{-}ck}(v) \\ & & \mid & \mathsf{set\text{-}ck}(v, v') \mid \mathsf{xhr}(v, v') \mid v \\ \textit{Scripts} & scr & ::= & \lceil e \rfloor^{\tau}_{@u} \end{array}$$

A script $\lceil e \rfloor^{\tau}_{@u}$ is an expression e running in the *origin* u with an associated tag τ. The origin u is needed to enforce the same-origin policy on accesses to the cookie jar, while the tag τ is used to enforce micro-policies on the script.

The expression $(\lambda x.e)\, v$ evaluates to $e\{v/x\}$; the expression $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ first evaluates e to a value v and then behaves as $e'\{v/x\}$; the expression $\mathsf{get\text{-}ck}(k)$ returns the value of the cookie with key k, provided that the tag assigned to the cookie allows this operation; the expression $\mathsf{set\text{-}ck}(k, v)$ stores the cookie $\mathsf{ck}(k, v)^{\tau}$ in the cookie jar, where $\tau = \kappa(\mathit{host}(u), k)$ is a tag derived by the origin u in which the expression is running and the cookie key k; again, the setting operation may fail due to the enforcement of a micro-policy. The expression $\mathsf{xhr}(u, \lambda x.e)$ sends an AJAX request to u and, when a value v is available as a response, it runs $e\{v/x\}$ in the same origin of the script which sent the request. Notably, micro-policies may also be used to constrain AJAX communication in FF^{τ}. For simplicity, in our model we assimilate to AJAX requests any network request which may be triggered by a script, e.g., the request for an image triggered by the insertion of a markup element in the page where the script is running.

Events

Input events i are defined as follows:

$$\begin{array}{lll} i & ::= & \mathsf{load}(u) \\ & \mid & \mathsf{doc\_resp}_n(u : CK, e) \mid \mathsf{doc\_redir}_n(u : CK, u') \\ & \mid & \mathsf{xhr\_resp}_n(u : CK, v) \mid \mathsf{xhr\_redir}_n(u : CK, u'). \end{array}$$

The event $\mathsf{load}(u)$ models a user navigating the browser to the URL u: the browser opens a new network connection to u, sends a HTTP(S) request and then waits for a corresponding HTTP(S) response to process over the connection. The event $\mathsf{doc\_resp}_n(u : CK, e)$ represents the reception of a document response from u, including a set of cookies CK to set and an expression e to run in the origin u, which leads to the execution of a new script. The event is annotated with the name n of the network connection where the response is received: this connection gets closed when processing the event. The event $\mathsf{doc\_redir}_n(u : CK, u')$ models the reception of a HTTP(S) redirection from u to u' along the connection n, setting the set of cookies CK; the event keeps the connection open, while pointing it to u'. A similar intuition applies to XHR responses and redirects. For simplicity, we use $\mathsf{net\_resp}_n(u : CK, e)$ to stand for any network response, including redirects.

Output events o are defined as follows:

$$o \quad ::= \quad \bullet \mid \mathsf{doc\_req}(u : CK) \mid \mathsf{xhr\_req}(u : CK).$$

The event $\bullet$ represents a silent reaction to an input event with no visible side-effect. The event $\mathsf{doc\_req}(u : CK)$ models a document request sent to u, including the set of cookies CK. The event $\mathsf{xhr\_req}(u : CK)$ models an XHR request sent to u, including the set of cookies CK. We let $\mathsf{net\_req}(u : CK)$ represent an arbitrary network request when we do not need to precisely identify its type.

States

Browser states are tuples $Q = \langle K, N, H, T, O \rangle$:

$$\begin{array}{llll} \textit{Cookie Jar} & K & ::= & \{\} \mid K \uplus \{d : CK\}, \\ \textit{Connections} & N & ::= & \{\} \mid N \uplus \{n^\tau : u\} \\ \textit{Handlers} & H & ::= & \{\} \mid H \uplus \{n^\tau : (u', \lceil \lambda x.e \rfloor_{@u})\}, \\ \textit{Tasks} & T & ::= & \mathsf{wait} \mid \mathit{scr}, \\ \textit{Outputs} & O & ::= & [\,] \mid o :: O'. \end{array}$$

The cookie jar K maps domain names to the cookies they set in the browser. The network connection store N keeps track of the pending document requests: if $\{n^\tau : u\} \in N$, then the browser is waiting for a document response from u over the connection n. Notice that the network connection includes a tag τ, which makes it possible to enforce micro-policies on it. The handler store H tracks pending XHR requests: if $H(n^\tau) = (u', \lceil \lambda x.e \rfloor_{@u})$, the continuation $\lambda x.e$ is ready to be run in the origin u when an XHR response is received from u' over the connection n. Also these connections have an associated tag.

We use T to represent *tasks*: if $T = \lceil e \rfloor^\tau_{@u}$, then a script is running; if $T = \mathsf{wait}$, no script is running. Finally, O is a buffer of output events, needed to interpret FF^τ as a reactive system: let $Q = \langle K, N, H, T, O \rangle$ be a consumer state when $T = \mathsf{wait}$ and $O = [\,]$, otherwise let Q be a producer state. We let $C_0 = \langle \{\}, \{\}, \{\}, \mathsf{wait}, [\,] \rangle$ be the initial state of FF^τ.

4.4.2 Reactive Semantics

The reactive semantics of FF^τ is parametric with respect to a partial function *transfer* [dADG+15], which is roughly a tag-based security monitor operating on the browser model. The transfer function we consider has the following format:

$$\mathit{transfer}(\mathsf{event\_type}, \tau_1, \tau_2) = (\tau_n, \tau_{ci}, \tau_{co}, \tau_s),$$

where τ_1 and τ_2 are the (at most two) arguments passed to the function when the browser model processes an event of type $\mathsf{event\_type}$, while $\tau_n, \tau_{ci}, \tau_{co}, \tau_s$ are the (at most four) tags assigned to the new browser elements which are instantiated as the result of the event processing. Specifically, τ_n is the tag of the new network connection which is created, τ_{ci} is the tag passed to the cookie jar when storing some new cookies, τ_{co} is the tag passed to the cookie jar when retrieving the cookies to be attached to HTTP(S) requests, and τ_s is the tag of the new running script. If any of these elements is not needed when processing an event of a given type, e.g., since no new cookie is set, we replace it with a dash ($-$). If the transfer function is undefined for a given set of

arguments, an operation is not permitted. For space reasons, the full reactive semantics of FF^{τ} is given in Appendix A.1.1. Here, we just present the main ideas needed to understand the paper.

A set of transitions of the form $C \xrightarrow{i} P$ describes how the consumer state C reacts to the input event i by evolving into a producer state P. Conversely, a set of transitions of the form $P \xrightarrow{o} Q$ describes how a producer state P generates an output event o and evolves into another state Q. Most of the transitions invoke the transfer function before being fired, with the following intuitive semantics:

- *transfer*(load, $\tau_u, -) = (\tau_n, -, \tau_{co}, -)$: invoked when a URL u such that *tag*$(u) = \tau_u$ is loaded. The event creates a new network connection with tag τ_n and uses tag τ_{co} to access the cookie jar and retrieve the cookies to be attached to the document request sent to u;
- *transfer*(doc_resp, $\tau_n, -) = (-, \tau_{ci}, -, \tau_s)$: invoked when a document response is received over a network connection with tag τ_n. The event uses tag τ_{ci} to access the cookie jar and set the cookies received in the response, while tag τ_s is given to the new script which is executed as the result of processing the response;
- *transfer*(doc_redir, $\tau_n, \tau_u) = (\tau_m, \tau_{ci}, \tau_{co}, -)$: invoked when a document redirect is received over a network connection with tag τ_n, asking the browser to load a URL u such that *tag*$(u) = \tau_u$. As a result, the tag of the network connection is changed from τ_n to τ_m. The tags τ_{ci} and τ_{co} are used to access the cookie jar: specifically, τ_{ci} is used to set the cookies received along with the processed redirect, while τ_{co} is used to get the cookies to be sent to u upon redirection;
- *transfer*(xhr_resp, $\tau_n, -) = (-, \tau_{ci}, -, \tau_s)$: similar to the case for doc_resp, but for XHR responses;
- *transfer*(xhr_redir, $\tau_n, \tau_u) = (\tau_m, \tau_{ci}, \tau_{co}, -)$: similar to the case for doc_redir, but for XHR redirects;
- *transfer*(send, $\tau_s, \tau_u) = (\tau_n, -, \tau_{co}, -)$: invoked when a script with tag τ_s sends an XHR request to a URL u such that *tag*$(u) = \tau_u$. Tag τ_n is given to the new network connection which is opened by the script, while τ_{co} is used to access the cookie jar and get the cookies to be sent to u;
- *transfer*(get, $\tau_r, \tau_c) = (-, -, -, -)$: invoked when a cookie with tag τ_c is read from the cookie jar. Here, τ_r is the tag modelling the security assumptions about the reader: for instance, when cookies are fetched by the browser for inclusion in a HTTP(S) request to u, this tag could correspond to the protocol of u;
- *transfer*(set, $\tau_w, \tau_c) = (-, -, -, -)$: invoked when a cookie with tag τ_u is written into the cookie jar. Similarly to the previous case, τ_w is the tag modelling the security assumptions about the writer.

If the transfer function is undefined for a specific set of tags, the corresponding transitions $C \xrightarrow{i} P$ and $P \xrightarrow{o} Q$ just lead to a dummy producer state firing the dummy event $\bullet$.

4.5 Enforcing Reactive Non-Interference

The operational semantics of FF^τ is parametric with respect to an arbitrary set of tags and a transfer function. Here, we instantiate these parameters to show that FF^τ can enforce a useful security property, i.e., reactive non-interference.

4.5.1 Labels, Policies and Threat Model

We define different threat models for the Web in terms of labels from a pre-order $(\mathcal{L}, \sqsubseteq)$, as required by the definition of reactive non-interference. We start by introducing *simple labels*, which we use to express confidentiality and integrity policies. A simple label l is a (possibly empty) set of elements of the form $\mathsf{http}(d)$ or $\mathsf{https}(d)$ for some domain name $d \in \mathcal{D}$:

$$l ::= \emptyset \mid \{\mathsf{http}(d)\} \mid \{\mathsf{https}(d)\} \mid l \cup l.$$

Intuitively, simple labels define sets of endpoints which are allowed to read/write a given datum or to observe/produce a given event. A *label* $\ell = (l_C, l_I)$ is a pair of simple labels, combining confidentiality and integrity. We write $C(\ell)$ for l_C and $I(\ell)$ for l_I. We let $\ell \sqsubseteq \ell'$ iff $C(\ell) \subseteq C(\ell')$ and $I(\ell) \subseteq I(\ell')$. Simple labels form a bounded lattice under set inclusion, while labels form a bounded lattice under $\sqsubseteq$: the bottom and top elements are $\bot_s = \emptyset$, $\top_s = \{\mathsf{http}(d), \mathsf{https}(d) \mid d \in \mathcal{D}\}$, $\bot = (\bot_s, \bot_s)$ and $\top = (\top_s, \top_s)$.

We assign (simple) labels to URLs, so that it is easy to define which events an attacker can observe and/or corrupt. For a URL $u \in \mathcal{U}$ with $\mathit{host}(u) = d$, we let:

- $\mathit{msg\_label}(u) = \{\mathsf{http}(d)\}$ iff $\mathit{prot}(u) = \mathsf{http}$;
- $\mathit{msg\_label}(u) = \{\mathsf{https}(d)\}$ iff $\mathit{prot}(u) = \mathsf{https}$;
- $\mathit{evt\_label}(u) = \{\mathsf{http}(d)\}$.

We use these functions to define the capabilities of an attacker ℓ. The *presence* of a message sent to u is visible to ℓ whenever $\mathit{evt\_label}(u) \subseteq C(\ell)$, while the *content* of the message is only disclosed if also $\mathit{msg\_label}(u) \subseteq C(\ell)$. If $\mathit{evt\_label}(u) \subseteq C(\ell)$ while $\mathit{msg\_label}(u) \not\subseteq C(\ell)$, the attacker is aware of the presence of all messages sent to u, but he has no access to their contents; the presence of a message may be used to create a side-channel and leak information through an implicit flow. As to integrity, a message coming from u can be *forged* by an attacker ℓ if and only if $\mathit{msg\_label}(u) \subseteq I(\ell)$. There is no distinction between message presence and message content when it comes to integrity.

Based on this informal description, the following well-formation hypothesis we make on the attacker should be clear. It ensures that we cannot model attackers who are not aware of the presence of a message, but still have access to its contents.

Definition 5 (Well-formed Attacker)**.** *An attacker ℓ is* well-formed *if and only if, for all domains $d \in \mathcal{D}$, $\mathsf{https}(d) \in C(\ell)$ implies $\mathsf{http}(d) \in C(\ell)$.*

Visibility of outputs:

$$\frac{\mathit{evt\_label}(u) \subseteq C(\ell)}{\mathit{vis}_\ell(\mathsf{net\_req}(u : CK))}$$

Indistinguishability of outputs:

$$\frac{\mathit{msg\_label}(u) \not\subseteq C(\ell)}{\mathsf{net\_req}(u : CK) \sim_\ell^C \mathsf{net\_req}(u : CK')}$$

Taintedness of inputs:

$$\frac{\mathit{msg\_label}(u) \subseteq I(\ell)}{\mathit{tnt}_\ell(\mathsf{net\_resp}_n(u : CK, e))}$$

Table 4.1: Attacker capabilities for ℓ

From now on, we always implicitly consider only well-formed attackers. It is easy to represent using labels several popular web security attackers:

1. a *web attacker* on domain d is defined by:

$$\ell_w(d) \triangleq (\{\mathsf{http}(d), \mathsf{https}(d)\}, \{\mathsf{http}(d), \mathsf{https}(d)\})$$

2. a *passive network attacker* is defined by:

$$\ell_{pn} \triangleq (\{\mathsf{http}(d) \mid d \in \mathcal{D}\}, \emptyset)$$

3. an *active network attacker* is defined by:

$$\ell_{an} \triangleq (\{\mathsf{http}(d) \mid d \in \mathcal{D}\}, \{\mathsf{http}(d) \mid d \in \mathcal{D}\}).$$

To formalize the previous intuitions, we introduce a few simple ingredients: a *visibility* predicate vis_ℓ on output events, a binary *indistinguishability* relation $\sim_\ell^C$ on output events, and a *taintedness* predicate tnt_ℓ on input events. These are defined in Table 4.1. The indistinguishability relation identifies network requests which only differ for contents (cookies) which are not visible to the attacker, because encrypted. We implicitly assume that two indistinguishable requests have the same type.

We then define two specific classes of non-interference policies, which correctly capture the attacker capabilities we described. Our non-interference results will be restricted to these two classes of policies.

Definition 6 (Confidentiality Policy)**.** *A* confidentiality policy *is a pair* $\pi_C = \langle \mathit{rel}_\ell, \sim_\ell \rangle$ *such that:*

1. $\forall o \in \mathcal{O} : \mathit{rel}_\ell(o) \triangleq \mathit{vis}_\ell(o)$;
2. $\forall o, o' \in \mathcal{O} : o \sim_\ell o' \Leftrightarrow o = o' \vee o \sim_\ell^C o'$.

Definition 7 (Integrity Policy)**.** *An* integrity policy *is a pair* $\pi_I = \langle \mathit{rel}_\ell, \sim_\ell \rangle$ *such that:*

1. $\forall i \in \mathcal{I} : \mathit{rel}_\ell(i) \triangleq \neg \mathit{tnt}_\ell(i)$;
2. $\forall i, i' \in \mathcal{I} : i \sim_\ell i' \Leftrightarrow i = i'$.

4.5.2 A Canonic Transfer Function for Non-Interference

Our goal is to instantiate the operational semantics of FF^τ with a transfer function that enforces confidentiality and integrity policies. In principle, we could let web developers provide selected entries of the transfer function, defining the browser behaviour upon interaction with their own websites, but this would be quite inconvenient for them. We believe that web developers need a more effective and declarative way to specify their desired confidentiality and integrity policies. In our view, web developers should only:

1. assign security labels to the cookies they set. This is not a hard task, since web developers are already familiar with cookie security attributes like HttpOnly and Secure, and the format of the labels is pretty intuitive;
2. assign security labels to the URLs they control. We argue that also this is not hard to understand for web developers, since this kind of policies is close in spirit to standard Content Security Policy [WBV15] specifications.

These labels define the expected security properties for cookies and network connections:

1. *cookie secrecy*: if a cookie has label ℓ, its value can only be disclosed to an attacker ℓ' such that $C(\ell) \cap C(\ell') \neq \emptyset$;
2. *cookie integrity*: if a cookie has label ℓ, it can only be set or modified by an attacker ℓ' such that $I(\ell) \cap I(\ell') \neq \emptyset$;
3. *session confidentiality*: if a URL u has label ℓ, an attacker ℓ' can observe that the browser is loading u only if we have $C(\ell) \cap C(\ell') \neq \emptyset$;
4. *session integrity*: if a URL u has label ℓ, an attacker ℓ' can force the browser into sending requests to u only if we have $I(\ell) \cap I(\ell') \neq \emptyset$.

Formally, we define a *URL labelling* as a function $\Gamma : \mathcal{U} \to \mathcal{L}$, assigning labels to URLs. If $\Gamma(u) = \ell$ for some label ℓ, let $\Gamma_C(u)$ stand for $C(\ell)$ and $\Gamma_I(u)$ stand for $I(\ell)$. We propose a technique to automatically generate a *canonic* transfer function from a labelling Γ: this function enforces the session confidentiality and integrity properties formalized by Γ, while ensuring that

cookies are accessed correctly according to their security label. The canonic transfer function operates on the set of tags *Tags* $\triangleq \mathcal{L} \cup \mathcal{U}$ including labels and URLs, and assumes that the tagging function for URLs *tag* is the identity on $\mathcal{U}$.

The definition of the canonic transfer function is formalized by using judgements of the following format:

$$\Gamma, f \rhd \mathit{transfer}(\mathsf{event\_type}, \tau_1, \tau_2) \rightsquigarrow \vec{\ell},$$

where Γ, f, event_type and τ_1, τ_2 are known, while we compute the labels $\vec{\ell}$ to be assigned to the newly created or updated browser elements when processing an event of type event_type. Here, $f : \mathcal{L} \to \mathcal{L}$ is a *script labelling*, providing a mapping from labels of network connections to labels of scripts downloaded via these connections. Remarkably, while Γ is used to specify different security policies for different URLs and should be provided by web developers, f is just a parameter used to tweak the generation of the canonic transfer function for different use cases. It is useful to have f in the formalism for additional generality, in particular to support the examples in the next section, but in practice (and in our implementation) a good candidate for f is simply the identity function on $\mathcal{L}$. The non-interference results we present hold for any choice of f, as long as it satisfies the following well-formation condition (implicitly assumed from now on).

Definition 8 (Well-formed Script Labelling)**.** *A script labelling f is* well-formed *if and only if, for all labels $\ell \in \mathcal{L}$, we have $C(f(\ell)) \subseteq C(\ell)$ and $I(\ell) \subseteq I(f(\ell))$.*

This well-formation requirement ensures that the confidentiality of a script is always higher than the confidentiality of the network connection from which it is downloaded, while its integrity is always lower than the integrity of the connection. This guarantees that the script cannot disclose the presence of private network connections or trigger high integrity events as the result of an interaction with the attacker.

The judgements defining the canonic transfer function are shown in Table 4.2, using inference rules which should be read as follows: boxed premises amount to checks on Γ, τ_1, τ_2, determining the domain of the transfer function, while premises not included in boxes define the value of the new labels $\vec{\ell}$. If any of the boxed premises fails, the transfer function is undefined and the browser does not process the event. Observe that the necessary entries of the transfer function can be generated "on the fly" upon event processing in an implementation of our theory.

We briefly comment on the rules in Table 4.2 as follows. In (G-Load), assuming that we load the URL u, we check $\mathit{evt\_label}(u) \subseteq \Gamma_C(u)$, since a request is sent to u and this request is visible to any network attacker or to any web attacker controlling u. We use $\Gamma_C(u)$ as the confidentiality label of the new network connection to ensure that the presence of that connection in the browser may only be visible to an attacker ℓ such that $\Gamma_C(u) \cap C(\ell) \neq \emptyset$. We use $\mathit{msg\_label}(u)$ as the integrity label of the new network connection, since an attacker controlling u may be able to compromise the integrity of any response received over that connection. Finally, when accessing the cookie jar to retrieve the cookies to be sent in the request to u, we set $\mathit{msg\_label}(u)$ as the confidentiality component of the label ℓ_{co} passed to the cookie jar, which ensures that only

(G-LOAD)
$$\frac{\boxed{\mathit{evt\_label}(u) \subseteq \Gamma_C(u)} \quad \ell_n = (\Gamma_C(u), \mathit{msg\_label}(u)) \quad \ell_{co} = (\mathit{msg\_label}(u), \top_s)}{\Gamma, f \rhd \mathit{transfer}(\mathsf{load}, u, -) \rightsquigarrow (\ell_n, -, \ell_{co}, -)}$$

(G-DOCRESP)
$$\Gamma, f \rhd \mathit{transfer}(\mathsf{doc\_resp}, \ell_n, -) \rightsquigarrow (-, \ell_n, -, f(\ell_n))$$

(G-DOCREDIR)
$$\frac{\boxed{\mathit{evt\_label}(u) \subseteq C(\ell_n)} \quad \boxed{I(\ell_n) \subseteq \Gamma_I(u)} \quad \ell_m = (C(\ell_n), I(\ell_n) \cup \mathit{msg\_label}(u)) \quad \ell_{co} = (\mathit{msg\_label}(u), \top_s)}{\Gamma, f \rhd \mathit{transfer}(\mathsf{doc\_redir}, \ell_n, u) \rightsquigarrow (\ell_m, \ell_n, \ell_{co}, -)}$$

(G-XHRRESP)
$$\Gamma, f \rhd \mathit{transfer}(\mathsf{xhr\_resp}, \ell_n, -) \rightsquigarrow (-, \ell_n, -, f(\ell_n))$$

(G-XHRREDIR)
$$\frac{\boxed{\mathit{evt\_label}(u) \subseteq C(\ell_n)} \quad \boxed{I(\ell_n) \subseteq \Gamma_I(u)} \quad \ell_m = (C(\ell_n), I(\ell_n) \cup \mathit{msg\_label}(u)) \quad \ell_{co} = (\mathit{msg\_label}(u), \top_s)}{\Gamma, f \rhd \mathit{transfer}(\mathsf{xhr\_redir}, \ell_n, u) \rightsquigarrow (\ell_m, \ell_n, \ell_{co}, -)}$$

(G-GET)
$$\frac{\boxed{C(\ell_r) \subseteq C(\ell_t)} \quad \boxed{I(\ell_t) \subseteq I(\ell_r)}}{\Gamma, f \rhd \mathit{transfer}(\mathsf{get}, \ell_r, \ell_t) \rightsquigarrow (-, -, -, -)}$$

(G-SET)
$$\frac{\boxed{C(\ell_t) \subseteq C(\ell_w)} \quad \boxed{I(\ell_w) \subseteq I(\ell_t)}}{\Gamma, f \rhd \mathit{transfer}(\mathsf{set}, \ell_w, \ell_t) \rightsquigarrow (-, -, -, -)}$$

(G-SEND)
$$\frac{\boxed{\mathit{evt\_label}(u) \subseteq C(\ell_s)} \quad \boxed{I(\ell_s) \subseteq \Gamma_I(u)} \quad \ell_n = (C(\ell_s), I(\ell_s) \cup \mathit{msg\_label}(u)) \quad \ell_{co} = (\mathit{msg\_label}(u), \top_s)}{\Gamma, f \rhd \mathit{transfer}(\mathsf{send}, \ell_s, u) \rightsquigarrow (\ell_n, -, \ell_{co}, -)}$$

Table 4.2: Generation of a canonic transfer function from Γ

cookies intended to be disclosed to u will be retrieved. We use $\top_s$ as integrity label for retrieving cookies, so that we get cookies irrespective of their integrity label.

(G-DocResp) and (G-XhrResp) propagate the label ℓ_n of the network connection to the cookie jar when setting new cookies included in a network response received over that connection. In terms of integrity, this implies that a network connection can only set cookies with lower integrity than itself. More subtly, in terms of confidentiality, this also implies that a network connection can only set cookies with higher confidentiality than itself: this is needed to ensure that the attacker cannot detect the occurrence of private network responses from the value (or the existence) of

public cookies set in those responses. The rules assign the label $f(\ell_n)$ to the new scripts running after response processing; here, the well-formation of f is crucial to ensure that the confidentiality and integrity restrictions of the script are as least as strong as those of the network connection where they have been downloaded.

In (G-DocRedir) and (G-XhrRedir), when redirecting to a URL u, we check $\mathit{evt\_label}(u) \subseteq C(\ell_n)$, since a network request is sent to u upon redirection and it may reveal the existence of the network connection. We also have to check $I(\ell_n) \subseteq \Gamma_I(u)$ to ensure that no low integrity connection sends a request to a high integrity URL. We preserve the confidentiality label of the existing network connection, so that the existence of the connection cannot be revealed even after the redirection. Instead, we update the integrity label of the connection to the original integrity label of the connection extended with $\mathit{msg\_label}(u)$: this formalizes the intuition that the integrity of a network connection gets downgraded through cross-origin redirects. The label used for writing cookies is ℓ_n, just as in (G-DocResp), while the label used for fetching cookies is $(\mathit{msg\_label}(u), \top_s)$, just as in (G-Load).

(G-Get) ensures that no high confidentiality cookie is read by a low confidentiality context and that no low integrity cookie is read by a high integrity context. (G-Set) is the writing counterpart of (G-Get). Finally, (G-Send) is similar to (G-XhrRedir), with the role of the incoming network connection taken by a running script (and no cookie set).

4.5.3 Reactive Non-Interference

Having defined a canonic transfer function, we now analyze which non-interference properties are supported by it. Let $\mathsf{FF}^{\tau}(\Gamma, f)$ be the instantiation of FF^{τ} with the transfer function derived from Γ and f using the judgements in Table 4.2.

We first discuss confidentiality. The next definition of erasure removes from input events any cookie which must not be visible to the attacker according to its label. By making similar input events that are identical after such erasure, reactive non-interference ensures that the value (and even the presence) of the confidential cookies has no visible import to the attacker.

Definition 9 (Confidentiality Erasure)**.** *Given a set of cookies CK, let ck-erase$^C_\ell(CK)$ be defined as:*

$$\{\mathsf{ck}(s, s')^{\ell'} \in CK \mid C(\ell') \cap C(\ell) \neq \emptyset\}.$$

We then define erase$^C_\ell : \mathcal{I} \to \mathcal{I}$ by applying ck-erase$^C_\ell$ to each CK syntactically occurring in an input event.

The confidentiality theorem combines cookie confidentiality with session confidentiality, i.e., the occurrence of a $\mathsf{load}(u)$ event which must not be visible to the attacker according to the label of u has indeed no visible import on the outputs produced by the browser.

Theorem 1 (Confidentiality)**.** *Let $\pi_C = \langle \mathit{rel}_\ell, \sim_\ell \rangle$ be the confidentiality policy such that:*

1. $\forall i : \neg \mathit{rel}_\ell(i) \triangleq i = \mathsf{load}(u) \wedge \Gamma_C(u) \cap C(\ell) = \emptyset$;

2. $\forall i, i' : i \sim_\ell i' \Leftrightarrow \mathit{erase}^C_\ell(i) = \mathit{erase}^C_\ell(i')$.

Then, $\mathsf{FF}^\tau(\Gamma, f)$ is non-interferent under π_C.

We now focus on integrity. The next definition of erasure removes from output events any cookie which can be set by the attacker according to its label. By making similar output events that are identical after such erasure, reactive non-interference ensures that only low-integrity cookies can be affected by a manipulation of the input stream performed by the attacker.

Definition 10 (Integrity Erasure)**.** *Given a set of cookies CK, let ck-erase$^I_\ell(CK)$ be defined as:*

$$\{\mathsf{ck}(s, s')^{\ell'} \in CK \mid I(\ell') \cap I(\ell) = \emptyset\}.$$

We then define $\mathit{erase}^I_\ell : \mathcal{O} \to \mathcal{O}$ by applying ck-erase$^I_\ell$ to each CK syntactically occurring in an output event.

The integrity theorem combines cookie integrity with session integrity, i.e., the attacker can force the browser into sending network requests to u only if the label of u has a low integrity component.

Theorem 2 (Integrity)**.** *Let $\pi_I = \langle \mathit{rel}_\ell, \sim_\ell \rangle$ be the integrity policy such that:*

1. $\forall o : \mathit{rel}_\ell(o) \triangleq o = \mathsf{net\_req}(u : CK) \wedge \Gamma_I(u) \cap I(\ell) = \emptyset$;
2. $\forall o, o' : o \sim_\ell o' \Leftrightarrow \mathit{erase}^I_\ell(o) = \mathit{erase}^I_\ell(o')$.

Then, $\mathsf{FF}^\tau(\Gamma, f)$ is non-interferent under π_I.

4.5.4 Proof Sketch

To prove the main theorems in the previous section, we first define a set of syntactic constraints over the structure of the transfer function aimed at enforcing non-interference. One example is the following constraint for load events:

$$\text{(T-LOAD)}\quad \frac{\begin{array}{c}\mathit{evt\_label}(u) \cup C(\ell_n) \subseteq \Gamma_C(u) \\ \mathit{msg\_label}(u) \subseteq C(\ell_{co}) \qquad \mathit{msg\_label}(u) \subseteq I(\ell_n)\end{array}}{\Gamma \vdash \mathit{transfer}(\mathsf{load}, u, -) = (\ell_n, -, \ell_{co}, -)}$$

Intuitively, rule (T-Load) ensures that, when a URL u is loaded, the information $\Gamma_C(u)$ is an upper bound for both $\mathit{evt\_label}(u)$ and the confidentiality label $C(\ell_n)$ of the new network connection. Having $\mathit{evt\_label}(u) \subseteq \Gamma_C(u)$ implies that the load event is always visible to any network attacker or any web attacker sitting at $\mathit{host}(u)$, while having $C(\ell_n) \subseteq \Gamma_C(u)$ guarantees that the side-effects produced by a response received over the network connection are only visible to $\Gamma_C(u)$. The rule also checks two other conditions: $\mathit{msg\_label}(u) \subseteq C(\ell_{co})$ is needed to ensure

that the cookies attached to the document request sent to u can actually be disclosed to it, while $msg\_label(u) \subseteq I(\ell_n)$ formalizes that an attacker who controls u may be able to compromise the integrity of any response received over the new network connection.

Having defined the full set of constraints, we then use a result from [BPS+09] to prove that the FF^{τ} model satisfies non-interference whenever it deploys a transfer function respecting the constraints.

Definition 11 (Unwinding Relation [BPS+09])**.** *An* unwinding relation *is a label-indexed family of binary relations $\mathcal{R}_\ell$ on states of a reactive system with the following properties:*

1. *if $Q\,\mathcal{R}_\ell\,Q'$, then $Q'\,\mathcal{R}_\ell\,Q$;*
2. *if $C\,\mathcal{R}_\ell\,C'$ and $C \xrightarrow{i} P$ and $C' \xrightarrow{i'} P'$ and $i \sim_\ell i'$ with $rel_\ell(i)$ and $rel_\ell(i')$, then $P\,\mathcal{R}_\ell\,P'$;*
3. *if $C\,\mathcal{R}_\ell\,C'$ and $C \xrightarrow{i} P$ with $\neg rel_\ell(i)$, then $P\,\mathcal{R}_\ell\,C'$;*
4. *if $P\,\mathcal{R}_\ell\,C$ and $P \xrightarrow{o} Q$, then $\neg rel_\ell(o)$ and $Q\,\mathcal{R}_\ell\,C$;*
5. *if $P\,\mathcal{R}_\ell\,P'$, then either of the following conditions hold true:*
   a) *$P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ with $o \sim_\ell o'$ and $Q\,\mathcal{R}_\ell\,Q'$;*
   b) *$P \xrightarrow{o} Q$ with $\neg rel_\ell(o)$ and $Q\,\mathcal{R}_\ell\,P'$;*
   c) *$P' \xrightarrow{o'} Q'$ with $\neg rel_\ell(o')$ and $P\,\mathcal{R}_\ell\,Q'$.*

Theorem 3 ([BPS+09])**.** *Let C_0 be the initial state of a reactive system R. If $C_0\,\mathcal{R}_\ell\,C_0$ for some unwinding relation $\mathcal{R}$, then R satisfies non-interference.*

We then need to define a suitable unwinding relation to establish non-interference. For confidentiality, we propose a relation that requires equality of the two browsers on the low-confidentiality components, while for integrity we propose a relation that requires equality on high-integrity components. Assuming the aforementioned constraints on the transfer function are respected, we prove that the relations fulfil the conditions of Definition 11 and hence conclude non-interference by Theorem 3. We then show that the canonic transfer function we defined always satisfies the set of constraints, from which we derive our two main theorems.

Note that this approach allows one to syntactically prove non-interference also for transfer functions different from the canonic one, which is a useful and interesting result by itself, as it creates an easy way to show security guarantees of a policy encoded with a custom transfer function. The proofs and the full set of constraints can be found in Appendix A.1.2 and Appendix A.2.1 - A.2.3.

4.5.5 Compatibility and Precision

Another interesting property of the canonic transfer function is that it ensures *compatibility* for websites not implementing the security mechanisms proposed in this paper. Intuitively, it is possible to identify a "weak" URL labelling Γ which does not improve security with respect to standard web browsers, but ensures that no runtime security check performed by the transfer function will ever stop a website from working as originally intended. Formally, we extend the set of output events of FF^{τ} with a new event $\star$, called *failure*. We then define a variant of FF^{τ} which is parametric with respect to a URL labelling Γ and explicitly models failures due to the security enforcement performed by the canonic transfer function derived from Γ. This is done by including the event $\star$ in the output stream generated by the reactive system whenever the transfer function is undefined. The failure semantics is presented in Appendix A.1.3.

Let $\Gamma_{\top}$ be the URL labelling assigning the $\top$ label to each URL, let id be the identity function on labels and let $\mathsf{FF}^{\tau}_{\star}(\Gamma_{\top}, id)$ be the failure-aware variant of FF^{τ} implementing the canonic transfer function derived from $\Gamma_{\top}$ and id. We can state the following compatibility theorem.

Theorem 4 (Compatibility)**.** *Let C_0 be the initial state of $\mathsf{FF}^{\tau}_{\star}(\Gamma_{\top}, id)$ and assume that the function $\kappa : \mathcal{D} \times \mathcal{S} \to$ Tags assigns the top label $\top$ to all the elements of its domain. If $C_0(I) \Downarrow O$, then $\star$ does not occur in O.*

While we guarantee the soundness of our approach, we cannot offer perfect *precision*, meaning that our framework conservatively prevents some information flows, even though non-interference is not violated. This is because we do not analyse JavaScript and, consequently, we assume a worst case scenario where information leaks may happen. For example, a script may try to read a confidential cookie and then send an unrelated request to an untrusted domain, which would not break confidentiality. However, without an information flow analysis for JavaScript, this cannot be guaranteed, and thus our approach prevents either the access to the cookie or the network request.

4.6 Case Studies

4.6.1 Cookie Protection Against Web Attackers

The HttpOnly attribute has been proposed as an in-depth defense mechanism for authentication cookies against web attackers [Bar11]. If a cookie is marked as HttpOnly, the browser forbids any access to it by JavaScript, thus preventing its theft through a successful XSS exploitation. The HttpOnly attribute also provides some integrity guarantees, since JavaScript cannot set or overwrite HttpOnly cookies[1].

Intuitively, a first attempt at representing HttpOnly cookies in our model can be done by giving cookies set by the domain d the label $\ell_c = (\{\mathsf{http}(d), \mathsf{https}(d)\}, \{\mathsf{http}(d), \mathsf{https}(d)\})$, and by

[1] Though this is not stated explicitly in the cookie specification [Bar11], this is a very sensible security practice and we experimentally verified it on many modern web browsers. It would be easy to model in our framework also HttpOnly cookies which can be set/overwritten by JavaScript, but we preferred to consider the more secure and common behaviour.

ensuring that scripts are assigned the top label $\top$. The label ℓ_c allows the browser to send and set these cookies over both HTTP and HTTPS connections to d. The label $\top$ assigned to scripts, instead, ensures that JavaScript cannot read or write these cookies, as enforced by rules (G-Get) and (G-Set).

As it turns out, however, this labelling forces the implementation of stricter security checks than those performed by standard web browsers on HttpOnly cookies. This is not a limitation of our model, but rather a consequence of the fact that scripts are actually able to compromise the integrity of HttpOnly cookies in current web browsers. Indeed, even if an attacker-controlled script cannot directly set an HttpOnly cookie by accessing the `document.cookie` property, it can still force HttpOnly cookies into the browser by exploiting network communication. For instance, assume that a trusted website `a.com` uses HttpOnly cookies for authentication purposes: a malicious script could run a login CSRF attack by submitting the attacker's credentials to `a.com`, thus effectively forcing fresh HttpOnly cookies into the user's browser.

To prevent this class of attacks, the canonic transfer function enforces two further invariants: (1) by rule (G-Send), all the network connections which are opened by a script are tagged with label $\top$, which enforces that no cookie with integrity label $\{\mathsf{http}(d), \mathsf{https}(d)\}$ can be set over these connections; and (2) by rules (G-DocRedir) and (G-XhrRedir), when a redirect is performed, the integrity label of the network connection receiving the redirect is downgraded to the union of the original integrity label and the message label of the redirect URL. Hence, if a cross-domain redirect is performed, no cookie with integrity label $\{\mathsf{http}(d), \mathsf{https}(d)\}$ can be set over the network connection. This ensures that web attackers cannot exploit malicious scripts or redirects to set cookies with label ℓ_c in the browser, unless they control the domain d.

In the end, standard HttpOnly cookies cannot be accurately modelled in our framework, since the integrity guarantees they provide cannot be expressed by a non-interference policy. Indeed, HttpOnly cookies cannot be set by a script using the `document.cookie` property, but scripts can still set them by exploiting network communication, so it is not clear which label should be assigned to scripts to have non-interference. As we discussed, this asymmetry leaves room for attacks.

Clearly, one can represent in our framework cookies which cannot be read by scripts, but can be set by them, by replacing the cookie label ℓ_c with $\ell'_c = (\{\mathsf{http}(d), \mathsf{https}(d)\}, \top_s)$. These cookies cannot be read by scripts running with the $\top$ label, but they can be liberally set by them. Another plausible design choice would be changing the label given to scripts to let them access cookies labelled ℓ'_c, at the cost of limiting their cross-origin communication. For instance, by giving scripts the label of the connection where they have been downloaded, scripts from the domain d would be allowed to read cookies labelled ℓ'_c, but any cross-domain communication from these scripts will be forbidden by rule (G-Send) to prevent cookie leakage. This may be a better solution for web applications like e-banking services, which may need to access session state at the client side, but do not interact with untrusted third-parties.

4.6.2 Protection Against Gadget Attackers

The gadget attacker has been first introduced in [BJM08b] as a realistic threat model for mashup security. A gadget attacker is just a web attacker with an additional capability: a trusted website deliberately embeds a gadget (script) chosen by the attacker as part of its standard functionalities. The embedded gadget may be useful, e.g., for advertisement purposes or for the computation of site-wide popularity metrics. It is well-known that this kind of operation is dangerous on the Web, since the embedded script may be entitled to run in the same origin of the embedding page [RKW12]. For instance, the embedded script may be able to read the authentication cookies of the embedding page. This is largely accepted, however, as long as the author of the embedding page trusts the gadget. But what if the gadget is compromised by the attacker?

Consider a web page hosted at the HTTPS URL u on domain d and loading a gadget from the HTTPS URL u' on domain d'. We can define a labelling Γ with $\Gamma_C(u) = \{\mathsf{https}(d), \mathsf{https}(d')\}$, which would allow the web page at u to only communicate with HTTPS URLs hosted at d and d' by rule (G-Send). This may be fine, for instance, if the gadget loaded from u' only computes some local statistics shown in the web page at u. Pick now the following input stream:

$$\begin{array}{rcl} I & = & [\mathsf{load}(u), \\ & & \mathsf{doc\_resp}_n(u : \{\mathsf{ck}(k, v)^\ell\}, \mathsf{xhr}(u', \lambda x.x\,\mathsf{unit})), \\ & & \mathsf{xhr\_resp}_m(u' : \emptyset, \lambda y.\mathsf{let}\ z = \mathsf{get\text{-}ck}(k)\ \mathsf{in}\ \mathit{leak}(z))] \end{array}$$

The input stream I models a scenario where the normally harmless gadget on u' has been somehow compromised by the attacker, so that it will read the value of the cookie k set by the response from u and leak it to the attacker's website, which we assume to be hosted outside the domains d and d'. This attack is prevented by the labelling above, since the XHR request leaking the cookie value is stopped by rule (G-Send), given that this request would still originate from u.

4.6.3 Strengthening PayPal

In 2014 a severe CSRF vulnerability on the online payment system PayPal was disclosed, despite existing server-side protection mechanisms [Ali14]. PayPal employs authentication tokens in order to prevent CSRF attacks, but these tokens could be used multiple times for the same user and, through another vulnerability, it was possible for an attacker to obtain such a valid token for any user. The combination of these two flaws could be used to mount arbitrary CSRF attacks against any PayPal user, e.g., to authorize payments on the user's behalf.

Figure 4.1 represents a typical payment scenario [RDJP11] for a user that has already logged into her PayPal account. The protocol starts with the user clicking on the "buy now" button in an online shop. After being redirected to PayPal, she confirms the payment and the article is successfully purchased. One important detail for the present discussion is that the initial request to PayPal (step 2) is explicitly triggered by the user (step 1) in this scenario. Our goal is to enforce a policy that prevents CSRF attacks against PayPal, while still allowing benign payments. This can be done by setting $\Gamma(u) = (\top_s, \{\mathsf{https}(\texttt{paypal.com})\})$ for all URLs u of PayPal, while letting $\Gamma(u') = \top$ for all URLs u' of the online shop (as we are only interested in protecting PayPal here).

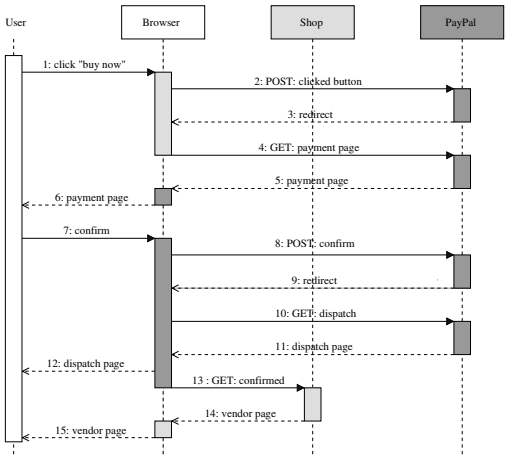


Figure 4.1: A typical payment scenario on PayPal

We first explain why this policy does not block benign payments via PayPal, like in Figure 4.1. Since we have an explicit user action that triggers the initial request to PayPal, we assimilate steps 1-2 of the protocol to the processing of a load event in our formal model. Since $\Gamma_C(u) = \top_s$ for all URLs u of PayPal, the request at step 2 is successfully sent according to rule (G-Load). Steps 3-12 of the protocol are always in the domain of PayPal and thus the label used for scripts and network connection is always $(\top_s, \{\mathsf{https}(\texttt{paypal.com})\})$. This allows the browser to perform arbitrary redirects and XHR request to URLs on PayPal, hence all these steps succeed. Finally, since we have $\Gamma_C(u) = \top_s$ for all URLs u at PayPal and we have $\Gamma_I(u') = \top_s$ for all URLs at the shop domain, also steps 13-15 can be performed successfully, since the cross-origin redirect is permitted by rule (G-DocRedir).

If we now consider a CSRF attack, then we do not have a message that is triggered directly by the user at step 1. This means that we do not have a load event to process, but rather a xhr_req, doc_redir or xhr_redir event coming from a domain different from `paypal.com`. By the definition of the canonic transfer function, we then always have to show either $I(\ell_n) \subseteq \Gamma_I(u)$ or $I(\ell_s) \subseteq \Gamma_I(u)$ in these cases, where u is the URL loaded at step 1. One can then observe that we always have $I(\ell_n) \not\subseteq \{\mathsf{https}(\texttt{paypal.com})\}$ and $I(\ell_s) \not\subseteq \{\mathsf{https}(\texttt{paypal.com})\}$, hence the integrity checks fail and the message is not sent. Thus, our technique effectively prevents the aforementioned CRSF attack against PayPal, even if vulnerabilities are not fixed at the server side.

4.6.4 Additional Examples

We also developed two more examples to show our framework at work: cookie protection against network attackers improving on Secure cookies [Bar11] and protection against CSRF in the spirit of Allowed Referrer Lists [CMKW13]. For space reasons, these examples are only included in Appendix A.1.4.

4.7 Implementation

We developed a proof-of-concept implementation of a significant core of our proposal as a Google Chrome extension, Michrome, which we make available online [CFGM16b]. We see Michrome as a first reasonable attempt at evaluating the practicality of our theory rather than as a finished product ready for inclusion in standard web browsers. More work is needed to support all the features of FF^{τ}: for instance, the current prototype lacks support for defining arbitrary cookie labels. We comment in the following on the main implementation choices, our experiments and the current limitations of Michrome.

4.7.1 Michrome Implementation

Michrome changes the behaviour of Google Chrome by mimicking the operational semantics of FF^{τ}, assuming the deployment of the canonic transfer function in Section 4.5.2. The prototype leverages the standard Google Chrome extension APIs, which allow for a rather direct implementation of the semantics. Michrome intercepts web requests via the APIs, and allows or denies them based on the underlying information flow policy. If a blocked request is a navigation to another page, the user receives a message that her request was blocked by the extension. If a blocked request is loading additional content for a page, the user does not receive any specific notification, but she might see differences in the page (e.g., missing images). We discuss below the main differences between Michrome and the presented formal model.

User Inputs

FF^{τ} uniformly treats user inputs as load events. In practice, however, users have different ways to interact with their web browser, most notably by typing in the address bar and by clicking buttons or links. Assimilating all user inputs to load events in Michrome would be a poor design choice, since many of these inputs, e.g., button clicks, can be triggered by malicious JavaScript code, but load events have high integrity in our model. Unfortunately, the Google Chrome extension APIs do not allow one to discriminate between user clicks and clicks performed by JavaScript; similarly, they do not provide any way to distinguish between the user writing in the address bar and a navigation attempt by JavaScript.

Our choice is then to only endorse the first request which is fired from an empty tab and to deem it as the result of a load event, since the only way to trigger a network request from an existing empty tab is by typing in its address bar. All the other network requests are assimilated to less trusted xhr_req events and hence subject to stricter security checks. This policy can be relaxed by defining in Michrome a white-list of trusted entry points, i.e., URLs which are known to be controlled by trusted companies and have a very high assurance of being protected against CSRF attacks: a similar approach has already been advocated in App Isolation [CBR+11]. Relaxing the standard behaviour of Michrome is occasionally helpful in practice, for instance to support the PayPal case study (see below).

Tagging Scripts

in normal web browsing, many scripts run in the same page (and hence in the same origin) at the same time. The Google Chrome extension APIs do not allow one to detect which script is performing a given operation when more than one script is included in the same page, so Michrome cannot assign labels to individual scripts (unlike the FF^{τ} model). This issue is solved by giving a label to the entire tab displaying the page rather than to individual scripts. Intuitively, this label represents an upper bound for each label which would be assigned to a script running in the tab.

When a remote content is included from a URL u, the label of the including tab is downgraded and joined with $\Gamma(u)$. There is only one simple exception to this rule: if the included content is passive, i.e., if it is an image, the integrity label of the tab does not get downgraded. This prevents label creeping for integrity and simplifies the specification of information flow policies for web developers.

Policy Granularity

In the current prototype, security labels are assigned to domain names rather than to URLs. This choice is mainly dictated by the practical need of testing our extension on existing websites: having a more coarse-grained security enforcement simplifies the process of writing information flow policies for websites we do not know and control. There is no real mismatch from the formal model here: we just implicitly assume that $\Gamma(u) = \Gamma(u')$ for all u, u' such that $\mathit{host}(u) = \mathit{host}(u')$.

Default Behaviour

Formally, the labelling Γ is defined as a (total) function from URLs to labels. In practice, however, one cannot assign a label to all the URLs in the Web. Our choice is to implicitly assume the $\top$ label for all the URLs without an explicit entry in Γ. This solution is suggested by the choice of preserving compatibility with existing websites: since the $\top$ label does not constrain cross-origin communication, the browser behaviour is unchanged when interacting with URLs not included in the labelling.

Cookies

At the time of writing, Michrome does not have full support for cookie labels. We plan to implement support for arbitrary labels in the next future, but the current prototype always assumes that a cookie received from a URL u has confidentiality label $\Gamma_C(u)$ and integrity label $\top_s$. This choice is mainly done for the sake of simplicity: by implicitly inferring cookie labels from Γ, we reduce the amount of information which we must specify for the websites we test. The confidentiality label $\Gamma_C(u)$ is justified by the fact that we assume that all URLs on the same domain have the same Γ, hence all the cookies set by them cannot be communicated outside

$\Gamma_C(u)$[2]. The integrity label $\top_s$, instead, is motivated by backward compatibility: since standard cookies do not provide good integrity guarantees, we do not try to enforce additional protection in order to avoid breaking websites.

4.7.2 Experiments

We performed a first test of Michrome by securing a university website, call it U. Since this website does not include many third-party contents and does not expect to process cross-domain requests, we first assigned U the label: $(\{\mathsf{http}(U), \mathsf{https}(U)\}, \{\mathsf{http}(U), \mathsf{https}(U)\})$. This label states that any session established with U should only be visible to U itself and that only local web pages are allowed to send requests to U. We then realized that this labelling modifies the browser behaviour when navigating U, since the homepage of U silently includes scripts from Google Analytics (*GA*) over HTTP and the extension blocks any request for these scripts, since *GA* should not be aware of the loading of U.

We tried to make Google Analytics work again by adding $\mathsf{http}(GA)$ to the confidentiality label of U. This indeed allowed the browser to send the request for the analytic scripts, but it also prevented the correct rendering and navigation of U later on. The reason is that, when a script from *GA* is included into a page on U, the integrity label of the tab displaying the page is downgraded to include $\mathsf{https}(GA)$. Since the integrity label of U does not mention *GA*, further requests to U from the page are dropped by Michrome: indeed, these requests may be fired by a malicious script mounting a CSRF attack. To recover functionality, we thus had to relax the integrity label of U to also include $\mathsf{https}(GA)$.

Another small change we had to perform to seamlessly navigate U was to extend its confidentiality label to include the sub-domain where the private area of the university is hosted. We also realised the need to include Google (G) in the integrity label of U, otherwise Michrome would prevent the browser from accessing U from the Google search page. Perhaps surprisingly, though Google is entirely deployed over HTTPS, extending the integrity label of U with just $\mathsf{https}(G)$ does not suffice to fully preserve functionality. The reason is that, just like most users, we often omit the protocol and just type `www.google.com` in the address bar to access Google: the browser then tries HTTP by default and then gets automatically redirected to HTTPS by Google. Hence, the integrity label of the tab after the redirect becomes $\{\mathsf{http}(G), \mathsf{https}(G)\}$, which is not good enough to access U. This problem can be solved by using HSTS [HJB12] and preventing any communication attempt to Google over HTTP.

An alternative, simpler solution to this problem is listing the home page of U as a trusted entry point in Michrome, so as to avoid listing all the most popular search engines in the integrity label of U. This is a safe choice in practice, since the homepage of U, like most homepages, is static and does not expect any parameter or untrusted input to sanitize. All in all, we found it pretty easy to come up with an accurate security policy for the website and we think that most web developers should find this process quite intuitive to carry out, especially since this whitelist-based approach is already advocated by existing web standards like Content Security Policy [WBV15].

[2]This is only true if no cookie sharing between sub-domains is possible. Indeed, the current prototype does not protect domain cookies [Bar11], but we plan to include support for them in future releases.

We also tested Michrome by placing an order on a well-known digital distribution platform, call it D, and by performing the payment using PayPal. We first built an entry for D in the URL labelling, ensuring that both the confidentiality and the integrity components of its label only included D and PayPal. We then set the confidentiality label of PayPal to $\top_s$ and its integrity label to PayPal itself (over HTTPS), thus reconstructing the scenario in the Section 4.6.3. The payment process worked seamlessly, confirming the result we expected from the formal model.

4.7.3 Compatibility and Perceived Performances

Besides the experiments detailed above, we also wrote information flow policies for a small set of national websites and we left Michrome activated in our web browsers while routinely browsing the Web for a few days. We never encountered any visible compatibility issue, even when interacting with websites without an explicit label in the URL labelling, which confirms that the $\top$ label given to them is a sensible default. Clearly, we occasionally broke websites when trying to come up with a correct label to assign to them, but this operation only needs to be done once per website (and only if additional protection is desired for that website). We envision a collaborative effort by security experts and web developers to write down policies for the major security-relevant websites, as it already happens for HTTPS Everywhere [Ele15].

We did not observe any perceivable performance degradation in any of the visited websites, which we do not find surprising, given that the security enforcement ultimately boils down to a few (light-weight) checks on labels.

4.7.4 Towards Full Practical Deployment

As we anticipated, Michrome is a proof-of-concept implementation of our approach intended for a first evaluation of its practicality. We implemented Michrome as a browser extension primarily for the sake of simplicity, since the Google Chrome extension APIs are powerful enough to allow us to implement a significant core of our formal framework with limited effort. We are currently investigating whether the entire proposal put forward in this paper can be securely implemented just by using a browser extension. This is not a trivial task, in particular there are (at least) two particularly interesting problems to address. First, implementing support for arbitrary cookie labels would require one to inject wrappers around the getters and setters of the `document.cookie` property. This can be done using a browser extension, but proving the security of the wrappers against arbitrary malicious scripts may be hard: we plan to study existing literature on language-based techniques for isolating JavaScript [BDM13, MT09] to address this issue. Moreover, we are investigating to which extent the security guarantees provided by Michrome may hold in presence of other extensions running in the browser: formal browser models representing the extension framework may be useful for the task [BCJ+15]. Understanding how effectively browser extensions can be employed for improving browser security is an interesting direction in general, since extensions are very easy to deploy and install, hence hold great promise for having a strong practical impact on web security.

4.8 Related Work

Browser-enforced security policies have already been proposed in the past, following two main lines of research. The first research line proposed *purely client-side* defenses such as ZAN [TDK11], SessionShield [NMY+11], CookiExt [BCFK14, BCFK15], CsFire [RDJP11] and SessInt [BCFK14], which automatically mitigate web applications vulnerabilities by changing the browser behaviour to prevent certain attacks. We improve over these works by giving web developers a tool to express their own browser-enforced security policies, using simple tools and abstractions. This choice makes our proposal more flexible and configurable than previous solutions. We argue that involving web developers in the security process is crucial for the usability and the large-scale deployment of a defensive solution, since purely client-side defenses like the ones we mentioned must implement heuristics to "guess" when their security policy should be applied. These heuristics are bound to (at least occasionally) fail: for instance, CookiExt sometimes breaks the Facebook chat [BCFK15], while CsFire prevents certain uses of the OpenId protocol [CMKW13].

The second research line on browser-side security, instead, focused on *hybrid* solutions similar to our approach, where the browser enforces a security policy specified by the server [JSH07, LV09, SSM10, WBS11, CMKW13]. These proposals, however, target very specific attacks like XSS [JSH07, LV09, SSM10] or CSRF [CMKW13], rather than providing full-fledged protection for web sessions. Also, these proposals have not been formalized and proved correct. Conversely, in this paper we formalize a rather general micro-policy framework for web browsers and we prove it is expressive enough to support a broad class of useful information flow policies, subsuming existing low-level security mechanisms for web sessions. We think that other useful security properties beyond non-interference can be enforced using micro-policies in the browser: we leave this study for future work.

The present paper was also inspired by previous work on information flow control for web browsers. FlowFox [GDNP12] was the first web browser enforcing a sound and precise information flow control on JavaScript by using secure multi-execution. There are many important differences between that approach and the one proposed in this paper. First, FlowFox exclusively prevents attacks posed by malicious scripts, while our proposal covers more common web threats, including malicious HTTP(S) redirects and network attacks. Second, FlowFox does not address integrity threats, though an extension explicitly aimed at thwarting CSRF attacks via scripts has been proposed [KCB+14]. Third, FlowFox requires profound changes to the JavaScript engine and has a quite significant impact on browsing performances, while we advocate a much more lightweight approach based on simple checks on labels. This is enough for the web session security properties we target. FlowFox, however, allows the specification of arbitrary fine-grained information flow policies on JavaScript which are beyond the scope of this work.

Fine-grained information flow control for web browsers, and JavaScript in particular, has also been proposed in [BRGH14, RBGH15]. These works extend a production JavaScript engine (WebKit) with dynamic information flow control operating at the level of bytecode: [BRGH14] presents a first implementation, extended in [RBGH15] to account for the intricacies of event handling and the DOM. Both the works come with a soundness proof, establishing termination-insensitive

non-interference for the enforcement mechanism. The relative strengths and weaknesses of our proposal with respect to [BRGH14, RBGH15] are essentially the same discussed in the comparison between our work and FlowFox. Combining browser-level micro-policies with fine-grained information flow control for JavaScript to provide precise, full-fledged protection for web sessions is an interesting research direction for future work.

Our proposal also shares similar design goals with coarse-grained information flow control frameworks for JavaScript like BFlow [YNKM09] and COWL [SYM+14]. These frameworks divide scripts in compartments and assign security labels to the latter, to then constrain communication across compartments based on label checks. An important difference with respect to these works is that the scope of the present paper is not limited to JavaScript. Moreover, we carry out our technical development in a formal model and prove security with respect to this model, while neither BFlow nor COWL have been formalized. Clearly, both BFlow and COWL support the enforcement of general information flow policies on JavaScript code, which is beyond the scope of the present work.

More recently, a research paper reported on the extension of Chromium with support for information flow control based on a lightweight, coarse-grained form of taint tracking [BCJ+15]. This proposal complements previous work on information flow control for JavaScript by focusing on the entire browser and embracing a wider range of web threats. It might be interesting to explore if the security mechanisms we advocate in this paper can be implemented using the security labels discussed in [BCJ+15]. The scope of the two works, however, is different: we focus on web session security, while [BCJ+15] targets intra-browser information flow policies. The threat model in [BCJ+15] is thus browser-centric, i.e., it identifies attackers with scripts and browser extensions; this is not enough for web session security, an area where network attackers must be taken into account. On the other hand, [BCJ+15] considers a more detailed browser model than the one used in this paper and it could be a good starting point to extend our work to deal with other threats, e.g., malicious browser extensions. Though we model a smaller fragment of the browser, our approach is intended to require way less changes to existing web browsers than the proposal in [BCJ+15]: indeed, our framework deliberately targets a good balance between strong web session security guarantees and minimal browser changes to simplify a practical adoption.

Finally, we observe that our label-based policies for confidentiality and integrity are reminiscent of the Same Origin Mutual Approval (SOMA) proposal [OWvOS08]. SOMA extends the browser with stricter access control checks on content inclusion: both the site operator of the including page and the third party content provider must approve a content inclusion before any communication is allowed by the browser. SOMA is shown to be effective in particular against CSRF attacks and malicious data exfiltration through XSS attacks, which are threats considered also in our work. There are two relevant differences, however, which make our proposal strictly more expressive than SOMA. First and most importantly, SOMA defines an access control mechanism and not an information flow framework: all the security checks performed by SOMA only depend on the including page and the embedded contents, and there is no way to allow or deny a content inclusion based on whether, e.g., the including page has been retrieved by a redirect from the attacker website. Second, SOMA only focuses on network communication and does not support security policies for cookies.

4.9 Conclusion

This work explores the usage of micro-policies for the specification and enforcement of confidentiality and integrity properties of web sessions. Micro-policies are specified in terms of tags (here, information flow labels) and a transfer function, which is responsible for monitoring security-relevant operations based on these tags. We modelled the browser as a reactive system and information flow security for web sessions as a non-interference property. We designed a synthesis technique for the transfer function, which allows the end user to specify the expected security policies as simple confidentiality and integrity labels. We demonstrated how our framework uniformly captures a broad spectrum of security policies (e.g., cookie protection, CSRF prevention, and gadget security), improving over existing ad-hoc solutions in terms of soundness and flexibility. We also managed to develop a proof-of-concept implementation of a significant core of our proposal as a simple and efficient Google Chrome extension, Michrome. Our experiments show that Michrome can be configured to enforce strong security policies without breaking the functionality of existing websites.

As future work, we plan to complete the implementation of Michrome to cover the entire framework presented in the paper. We also want to extend our formal model by considering additional browser and webpage components, striving for a good balance between formal expressiveness and ease of deployment in practice. We plan to formalize our development in a theorem prover in order to provide machine-checked security proofs. Furthermore, we would like to design micro-policies tailored to other popular web applications, such as single sign-on protocols, conducting a systematic security analysis of their deployment in the wild.

While performing experiments we realized that Michrome naturally acts as a *learning tool* that collects the integrity level of any web resource that is accessed by a web application. More specifically, when we do not assign security labels to a website, any access is allowed and the integrity level of the browser tab is populated by the security labels of the accessed URLs. This information is useful to have an immediate idea of the "trusted computing base" of the web application and, in many cases, to discover potential vulnerabilities such as importing scripts via HTTP. We plan to complement this learning feature with information about violations of the transfer function, so to automatically derive confidentiality and integrity labels for a whole web application.

CHAPTER 5

A Type System for Server Side Session Integrity

Abstract

Session management is a fundamental component of web applications: despite the apparent simplicity, correctly implementing web sessions is extremely tricky, as witnessed by the large number of existing attacks. This motivated the design of formal methods to rigorously reason about web session security which, however, are not supported at present by suitable automated verification techniques. In this paper we introduce the first security type system that enforces session security on a core model of web applications, focusing in particular on server-side code. We showcase the expressiveness of our type system by analyzing the session management logic of HotCRP, Moodle, and phpMyAdmin, unveiling novel security flaws that have been acknowledged by software developers.

The work presented in this chapter is the result of a collaboration with Stefano Calzavara, Riccardo Focardi, Matteo Maffei and Mauro Tempesta and was published at the 33rd IEEE Computer Security Foundations Symposium (CSF'20) under the title "Language-Based Web Session Integrity" [CFG+20]. I contributed to the design of the semantics and am responsible for the design of the type system and the formal proof presented in Appendix B.2. The formalization of real world examples has been performed by Mauro Tempesta.

5.1 Introduction

Since the HTTP protocol is stateless, web applications that need to keep track of state information over multiple HTTP requests have to implement custom logic for *session management*. Web sessions typically start with the submission of a login form from a web browser, where a registered user provides her access credentials to the web application. If these credentials are valid, the web

application stores in the user's browser fresh *session cookies*, which are automatically attached to all subsequent requests sent to the web application. These cookies contain enough information to authenticate the user and to keep track of session state across requests.

Session management is essential in the modern Web, yet it is often vulnerable to a range of attacks and surprisingly hard to get right. For instance, the theft of session cookies allows an attacker to impersonate the victim at the web application [NMY+11, BCFK15, TDK11], while the weak integrity guarantees offered by cookies allow subtle attacks like cookie forcing, where a user is forced into an attacker-controlled session via cookie overwriting [ZJL+15]. Other common attacks include cross-site request forgery (CSRF) [JKK06], where an attacker instruments the victim's browser to send forged authenticated requests to a target web application, and login CSRF, where the victim's browser is forced into the attacker's session by submitting a login form with the attacker's credentials [BJM08a]. We refer to a recent survey for an overview of attacks against web sessions and countermeasures [CFST17].

Given the complexity of session management and the range of threats to be faced on the web, a formal understanding of web session security and the design of automated verification techniques is an important research direction. Web sessions and their desired security properties have been formally studied in several papers developing browser-side defenses for web sessions [BCF+14, BCFK15, KCB+14, CFGM16a]: while the focus on browser-side protection mechanisms is appealing to protect users of vulnerable web applications, the deployment of these solutions is limited since it is hard to design browser-side defenses that do not cause compatibility issues on existing websites and are effective enough to be integrated in commercial browsers [CFST17].

Thus, security-conscious developers would better rely on server-side programming practices to enforce web session security when web applications are accessed by standard browsers. Recently, Fett et al. [FHK19] formalized a session integrity property specific to OpenID within the Web Infrastructure Model (WIM), an expressive web model within which proofs are, however, manual and require a strong expertise.

In this work, we present *the first static analysis technique for web session integrity*, focusing on sound server-side programming practices. In particular:

1. we introduce a core formal model of web systems, representing browsers, servers, and attackers who may mediate communications between them. Attackers can also interact with honest servers to establish their own sessions and host malicious content on compromised websites. The goal in the design of the model is to retain simplicity, to ease the presentation of the basic principles underlying our analysis technique, while being expressive enough to capture the salient aspects of session management in real-world case studies. In this model, we formalize a generic definition of *session integrity*, inspired by prior work on browser-side security [BCF+14], as a semantic hyperproperty [CS10] ruling out a wide range of attacks against web sessions;

2. we design a novel type system for the verification of session integrity within our model. The type system exploits confidentiality and integrity guarantees of session data to en-

dorse untrusted requests coming from the network and enforces appropriate browser-side invariants in the corresponding responses to guarantee session integrity;

3. we showcase the effectiveness and generality of our type system by analyzing the session management logic of HotCRP, Moodle, and phpMyAdmin. After encoding the relevant code fragments in our formal model, we use the type system to establish a session integrity proof: failures in this process led to the discovery of critical security flaws. We identified two vulnerabilities in HotCRP that allow an attacker to hijack accounts of authors and even reviewers, and one in phpMyAdmin, which has been assigned a CVE [MIT19]. All vulnerabilities have been reported and acknowledged by the application developers. We finally established security proofs for the fixed versions by typing.

5.2 Overview

In this Section we provide a high-level overview of our approach to the verification of session integrity. Full formal details and a complete security analysis of the HotCRP conference management system are presented in the remainder of the paper.

5.2.1 Encoding PHP Code in our Calculus

The first step of our approach consists in accessing the PHP implementation of HotCRP and carefully handcrafting a model of its authentication management mechanisms into the core calculus we use to model web application code. While several commands are standard, our language for server-side programs includes some high-level commands abstracting functionalities that are implemented in several lines of PHP code. The **login** command abstracts a snippet of code checking, e.g., in a database, whether the provided credentials match an existing user in the system. Command **auth** is a *security assertion* parametrized by expressions it depends on. In our encoding it abstracts code performing security-sensitive operations within the active session: here it models code handling paper submissions in HotCRP. Command **start** takes as argument a session identifier and corresponds to the `session_start` function of PHP, restoring variables set in the session memory during previous requests bound to that session.

In the following we distinguish standard PHP variables from those stored in the session memory (i.e., variables in the `$_SESSION` array) using symbols @ and $, respectively. The **reply** command models the server's response in a structured way by separating the page's DOM, scripts, and cookies set via HTTP headers.

5.2.2 A Core Model of HotCRP

We assume that the HotCRP installation is hosted at the domain d_C and accessible via two HTTPS endpoints: *login*, where users perform authentication using their access credentials, and *manage*, where users can upload their papers or withdraw their submissions. The session management logic is based on a cookie *sid* established upon login. We now discuss the functionality of the two HTTPS endpoints; we denote the names of cookies in square brackets and the name of

parameters in parentheses. The login endpoint expects a username uid and a password pwd used for authentication:

```
1. login[](uid, pwd) ↪
2.   if uid = ⊥ and pwd = ⊥ then
3.     reply ({auth ↦ form(login, ⟨⊥, ⊥⟩)}, skip, {})
4.   else
5.     @r := fresh(); login uid, pwd, @r;
6.     start @r; $user := uid;
7.     reply ({link ↦ form(manage, ⟨⊥, ⊥, ⊥⟩)},
8.             skip, {sid ↦ x}))
9.      with x = @r
```

If the user contacts the endpoint without providing access credentials, the endpoint replies with a page containing a login form expecting the username and password (lines 2–3). Otherwise, upon successful authentication via uid and pwd, the endpoint starts a new session indexed by a fresh identifier which is stored into the variable $@r$ (lines 5–6). For technical convenience, in the **login** command we also specify the fresh session identifier as a third parameter to bind the identity of its owner to the session. Next, the endpoint stores the user's identity in the session variable $\$user$ so that the session identifier can be used to authenticate the user in subsequent requests (line 6). Finally, the endpoint sends a reply to the user's browser which includes a link to the submission management interface and sets a cookie sid containing the session identifier stored in $@r$ (lines 7–9).

The submission management endpoint requires authentication, hence it expects a session cookie sid. It also expects three parameters: a $paper$, an $action$ (submit or withdraw) and a $token$ to protect against CSRF attacks [BJM08a]:

```
1. manage[sid](paper, action, token) ↪
2.   start @sid;
3.   if $user = ⊥ then
4.     reply ({auth ↦ form(login, ⟨⊥, ⊥⟩)}, skip, {})
5.   else if paper = ⊥ then
6.     $utoken = fresh();
7.     reply ({add ↦ form(manage, ⟨⊥, submit, x⟩),
8.             del ↦ form(manage, ⟨⊥, withdraw, x⟩)},
9.            skip, {})
10.     with x = $utoken
11.  else if tokenchk(token, $utoken) then
12.    auth paper, action at ℓC; reply ({}, skip, {})
```

The endpoint first tries to start a session over the cookie sid: if it identifies a valid session, session variables from previous requests are restored (line 2). The condition $\$user = \bot$ checks whether the session is authenticated, since the variable is only set after login: if it is not the case, the endpoint replies with a link to the login page (lines 3–4). If the user is authenticated but does not provide any paper in her request, the endpoint replies with two forms used to submit

or withdraw a paper respectively. Such forms are protected against CSRF with a fresh token, whose value is stored in the session variable $\$utoken$ (lines 5–10). If the user is authenticated and requests an action over a given paper, the endpoint checks that the token supplied in the request matches the one stored in the user's session (line 11) and performs the requested action upon success (line 12). This is modeled via a security assertion in the code that authorizes the requested action on the paper on behalf of the owner of the session. The assertion has a security label ℓ_C, intuitively meaning that authorization can be trusted unless the attacker can read or write at ℓ_C. Security labels have a confidentiality and an integrity component, expressing who can read and who can write. They are typically used in the information flow literature [CFGM16a] not only to represent the security of program terms but also the attacker itself. Here we let $\ell_C = (\mathsf{https}(d_C), \mathsf{https}(d_C))$, meaning that authorization can be trusted unless HTTPS communication with the domain d_C hosting HotCRP is compromised by the attacker.

5.2.3 Session Integrity

In this work, we are interested in *session integrity*. Inspired by [BCF+14], we formalize it as a relational property, comparing two different scenarios: an ideal world where the attacker does nothing and an attacked world where the attacker uses her capabilities to compromise the session. Intuitively, session integrity requires that any authorized action occurring in the attacked world can also happen in the ideal world, unless the attacker is powerful enough to void the security assertions; this must hold for all sequences of actions of a user interacting with the session using a standard web browser.

As a counterexample to session integrity for our HotCRP model, pick an attacker hosting an HTTPS website at the domain $d_E \neq d_C$, modeled by the security label $\ell_E = (\mathsf{https}(d_E), \mathsf{https}(d_E))$. Since $\ell_E \not\sqsubseteq \ell_C$, this attacker should not be able to interfere with authorized actions at the submission management endpoint. However, this does not hold due to the lack of CSRF protection on the endpoint *login*. In particular, pick the following sequence of user actions where *evil* stands for an HTTPS endpoint at d_E:

$$\begin{aligned}\vec{a} = \ & \mathsf{load}(1, \mathit{login}, \{\}),\\ & \mathsf{submit}(1, \mathit{login}, \texttt{auth}, \{1 \mapsto \mathsf{usr}, 2 \mapsto \texttt{pwd}\}),\\ & \mathsf{load}(2, \mathit{evil}, \{\}), \mathsf{submit}(1, \mathit{login}, \texttt{link}, \{\}),\\ & \mathsf{submit}(1, \mathit{manage}, \texttt{add}, \{1 \mapsto \texttt{paper}\})\end{aligned}$$

The user opens the login endpoint in tab 1 and submits her username and password via the authentication form (identified by the tag `auth`). She then loads the attacker's website in tab 2 and moves back to tab 1 where she accesses the submission management endpoint by clicking the link obtained upon authentication. Finally, she submits a paper via the `add` form.

Session integrity is violated since the attacker can reply with a page containing a script which automatically submits the attacker's credentials to the login endpoint, authenticating the user as the attacker at HotCRP. Thus, the last user action triggers the security assertion in the attacker's session rather than in the user's session. Formally, this is captured by the security assertion firing the event $\sharp[\texttt{paper}, \texttt{submit}]_{\ell_C}^{\mathsf{usr},\mathsf{atk}}$, modeling that the paper is submitted by the user into the

attacker's session. As such an event cannot be fired in the ideal world without the attacker, this violates session integrity.

In practice, an attacker could perform the attack against an author so that, upon uncareful submission, a paper is registered in the attacker's account, violating the paper's confidentiality. We also discovered a more severe attack allowing an attacker to log into the victim's session, explained in section 5.5.

5.2.4 Security by Typing

Our type system allows for sound verification of session integrity and is parametric with respect to an attacker label. In particular, typing ensures that the attacker has no way to forge authenticated events in the session of an honest user (as in a CSRF attack) or to force the user to perform actions within a session bound to the attacker's identity (e.g., due to a login CSRF). Failures arising during type-checking often highlight in a direct way session integrity flaws.

To ensure session integrity, we require two ingredients: first, we need to determine the identity of the sender of the request; second, we must ensure that the request is actually sent with the consent of the user, i.e., the browser is not sending the request as the attacker's deputy. Our type system captures these aspects using two labels: a *session* label and a *program counter* (PC) label. The session label models both the session's integrity (i.e., who can influence the session and its contents) and confidentiality (i.e., who can learn the session identifier used as access control token). Since the identity associated with an authenticated event is derived from the ongoing session, the session label captures the first ingredient. The PC label tracks who could have influenced the control flow to reach the current point of the execution. Since a CSRF attack is exactly a request of low integrity (as it is triggered by the attacker), this captures the second ingredient. Additionally, the type system relies on a typing environment that assigns types to URLs and their parameters, to local variables and to references in the server memory.

We type-check the code twice under different assumptions. First, we assume the scenario of an honest user regularly interacting with the page: here we assume that all URL parameters are typed according to the typing environment and we start with a high integrity PC label. Second, we assume the scenario of a CSRF attack where all URL parameters have low confidentiality and integrity (since they are controlled by the attacker) and we start with a low integrity PC label. In both cases, types for cookies and the server variables are taken from the typing environment since, even in a CSRF attack, cookies are taken from the cookie jar of the user's browser and the attacker has no direct access to the server memory.

We now explain on a high level why our type system fails to type-check our (vulnerable) HotCRP model. To type the security assertion $\mathbf{auth}\ paper, action\ \mathbf{at}\ \ell_C$ in the $manage$ endpoint, we need a high integrity PC label, a high integrity session label and we require the parameters $paper$ and $action$ to be of high integrity. While the types of the parameters are immediately determined by the typing environment, the other two labels are influenced by the typing derivation.

In the CSRF scenario, the security assertion is unreachable due to the presence of the token check instruction (line 11). When typing, if we assume (in the typing environment) that $\$ltoken$ is a

high confidentiality reference, we can conclude that the check always fails since the parameter *token* (controlled by the attacker) has low confidentiality, therefore we do not need to type-check the continuation.[1]

In the honest scenario, the PC label has high integrity assuming that all the preceding conditionals have high integrity guard expressions (lines 3 and 5). The session label is set in the command **start** @*sid* (line 2) and depends on the type of the session identifier @*sid*. To succeed in typing, @*sid* must have high integrity. However, we cannot type-check the *login* endpoint under this assumption: since the code does not contain any command that allows pruning the CSRF typing branch (like the token check in the *manage* endpoint), the entire code must be typed with a low integrity PC label. This prevents typing the **reply** statement where cookie *sid* is set (lines 7–9), since writing to a high integrity location from a low integrity context is unsound. In practice, this failure in typing uncovers the vulnerability in our code: the integrity of the session cookie is low since an attacker can use a login CSRF attack to set a session cookie in the user's browser.

As a fix, one can protect the *login* endpoint against CSRF attempts by using *pre-sessions* [BJM08a]: when the *login* endpoint is visited for the first time by the browser, it creates a new unauthenticated session at the server-side (using a fresh cookie *pre*) and generates a token which is saved into the session and embedded into the login form. When submitting the login form, the contained token is compared to the one stored at the server-side in the pre-session and, if there is a mismatch, authentication fails:

```
1. login[pre](uid, pwd, token) ↪
2.   if uid = ⊥ and pwd = ⊥ then
3.     @r′ := fresh(); start @r′; $ltoken := fresh();
4.     reply ({auth ↦ form(login, ⟨⊥, ⊥, x⟩)},
5.             skip, {pre ↦ y})
6.       with x = $ltoken, y = @r′
7.   else
8.     start @pre;
9.     if tokenchk(token, $ltoken) then
10.      @r := fresh(); login uid, pwd, @r;
11.      start @r; $user := uid;
12.      reply ({link ↦ form(manage, ⟨⊥, ⊥, ⊥⟩)},
13.              skip, {sid ↦ x})
14.        with x = @r
```

The session identified by *pre* has low integrity but high confidentiality: indeed, an attacker can cause a random *pre* cookie to be set in the user's browser (by forcing the browser to interact with the *login* endpoint), but she has no way to learn the value of the cookie and hence cannot access the session. We can thus assume high confidentiality for the session reference $*ltoken* in the session identified by *pre*.

[1] This reasoning is sound only when credentials (e.g., session identifiers and CSRF tokens) are unguessable fresh names. To take into account this aspect, in the type system we have special types for credentials (cf. subsection 5.4.1) and we forbid subtyping for high confidentiality credentials.

With the proposed fix, the piece of code responsible for setting the session cookie sid is protected by a token check, where the parameter $token$ is compared against the high confidentiality session reference $\$ltoken$ of the session identified by $@pre$ (line 9). Similar to the token check in the *manage* endpoint, this allows us to prune the CSRF typing branch and we can successfully type-check the code with a high integrity type for sid. We refer the reader to subsection 5.5.3 for a detailed explanation of typing the fixed $login$ endpoint.

The HotCRP developer acknowledged the login CSRF vulnerability and the effectiveness of the proposed fix, which is currently under development.

5.3 A Formal Model of Web Systems

We present now our model of web systems that includes the relevant ingredients for modeling attacks against session integrity and the corresponding defenses and we formally define our session integrity property.

5.3.1 Expressiveness of the Model

Our model of browsers supports cookies and a minimal client-side scripting language featuring *i)* read/write access to the cookie jar and the DOM of pages; *ii)* the possibility to send network requests towards arbitrary endpoints and include their contents as scripts. The latter capability is used to model resource inclusion and a simplified way to perform XHR requests. In the model we can encode many security-sensitive aspects of cookies that are relevant for attacks involving their theft or overwriting, i.e., cookie prefixes [Wes] and attributes `Domain` and `Secure` [Bar11]. We also model HSTS [HJB12] which can improve the integrity guarantees of cookies set by HSTS-enabled domains. On the server-side we include primitives used for session management and standard defenses against CSRF attacks, e.g., double submit cookies, validation of the `Origin` header and the use of CSRF tokens.

For the sake of presentation and simplicity, we intentionally omit some web components that are instead covered in other web models (e.g., the WIM [FHK19]) but are not fundamental for session integrity or for modelling our case studies. In particular, we do not model document frames and cross-frame communications via the Web Messaging API, web sockets, local storage, DNS and an equational theory for cryptographic primitives. We also exclude the `Referer` header since it conveys similar information to the `Origin` header which we already cover in our model. While we believe that our type system can be in principle extended to cover also these web components, the presentation and proof of soundness would become cumbersome, obfuscating the key aspects of our static analysis technique.

5.3.2 Syntax

We write $\vec{r} = \langle r_1, \ldots, r_m \rangle$ to denote a list of elements of length $m = |\vec{r}|$. We denote with r_k the k-th element of $\vec{r}$ and we let $r' :: \vec{r}$ be the list obtained by prepending the element r' to the list $\vec{r}$. A map M is a partial function from keys to values and we write $M(k) = v$ whenever the key k is bound to the value v in M. We let $dom(M)$ be the domain of M and $\{\}$ be the empty map.

Basics

Names	n^ℓ, i^ℓ, j^ℓ	$\in \mathcal{N}$		References	r	$\in \mathcal{R}$
Variables	x	$\in \mathcal{X}$		Identities	ι	$\in \mathcal{I} \ni \mathsf{usr}$
Domains	d	$\in \mathcal{D}$		URLs	u	$\in \mathcal{U}$
Origins	o	$\in \mathcal{O} \supseteq O$		Simple labels	l	$\in \mathcal{L} \supseteq L$
Labels	ℓ	$::= (l, l)$		Types	τ	$\in \mathcal{T}$
Primitive values	pv	$::= \mathit{true} \mid \mathit{false} \mid k \mid \ldots$		Numbers	k, m	$\in \mathbb{N}$
Values	v	$::= pv \mid n \mid \iota \mid u \mid \bot \in \mathcal{V}$		Metavariables	z	$\in \mathcal{V} \cup \mathcal{X}$
Forms	f	$::= \{\} \mid f \uplus \{v \mapsto \mathsf{form}(u, \vec{z})\}$		Pages	page	$::= \mathsf{error} \mid f$
Cookies	ck	$::= \{\} \mid ck \uplus \{r \mapsto z\}$		Memories	M	$::= \{\} \mid M \uplus \{r \mapsto v\}$

Servers

Expressions	se	$::= x \mid @r \mid \$r \mid v \mid \mathit{fresh}()^\ell \mid se \odot se'$
Environments	E	$::= i, \bot \mid i, j$
Request contexts	R	$::= n, u, \iota, l$
Databases	D	$::= \{\} \mid D \uplus \{n \mapsto M\}$
Trust mappings	ϕ	$::= \{\} \mid \phi \uplus \{n \mapsto \iota\}$
Servers	S	$::= (D, \phi, t)$
Threads	t	$::= u[\vec{r}](\vec{x}) \hookrightarrow c \mid \lceil c \rfloor_E^R \mid t \parallel t$
Commands	c	$::= \mathbf{skip} \mid \mathbf{halt} \mid c; c' \mid @r := se \mid \$r := se \mid \mathbf{if}\ se\ \mathbf{then}\ c\ \mathbf{else}\ c'$
		$\mid \mathbf{login}\ se_u, se_{pw}, se_{id} \mid \mathbf{start}\ se \mid \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell$
		$\mid \mathbf{if}\ \mathbf{tokenchk}(e, e')\ \mathbf{then}\ c \mid \mathbf{if}\ \mathbf{originchk}(L)\ \mathbf{then}\ c$
		$\mid \mathbf{reply}\ (\mathit{page}, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se} \mid \mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se}$

User behavior

Tab IDs	tab	$\in \mathbb{N}$
Inputs	p	$::= \{\} \mid p \uplus \{k \mapsto v^\tau\}$
Actions	a	$::= \mathsf{halt} \mid \mathsf{load}(tab, u, p) \mid \mathsf{submit}(tab, u, v, p)$

Web Systems

Attacker's Knowledge	$\mathcal{K}$	$\subseteq \mathcal{N}$
Web Systems	W	$::= B \mid S \mid W \parallel W$
Attacked Systems	A	$::= (\ell, \mathcal{K}) \triangleright W$

Table 5.1: Syntax (browsers B and scripts s are defined in Appendix B.1.1).

Given two maps M_1 and M_2, we define $M_1 \triangleleft M_2$ as the map M such that $M(k) = v$ iff either $M_2(k) = v$ or $k \notin \mathit{dom}(M_2)$ and $M_1(k) = v$. We write $M_1 \uplus M_2$ to denote $M_1 \triangleleft M_2$ if M_1 and M_2 are disjoint. We let $M\{k \mapsto v\}$ be the map obtained from M by substituting the value bound to k with v.

Basics

we let $\mathcal{N}$ be a set of names modeling secrets (e.g., passwords) and fresh identifiers that cannot be forged by an attacker. Names are annotated with a security label ℓ, that we omit in the semantics since it has no semantic effect. $\mathcal{R}$ is the set of references used to model cookies and memory locations, while $\mathcal{X}$ is the set of variables used for parameters and server commands. $\mathcal{I}$ is the set of identities representing users: we distinguish a special identity usr representing the honest user and we assume that the other identities are under the attacker's control.

A URL u is a triple (π, d, v) where $\pi \in \{\mathsf{http}, \mathsf{https}\}$ is the protocol identifier, d is the domain name, and v is a value encoding the path of the accessed resource. We ignore the port for the sake of simplicity. The origin of URL u is the simple label $\pi(d)$. For origins and URLs, we use $\perp$ for a blank value.

We let v range over values, i.e., names, primitive values (booleans, integers, etc.), URLs, identities and the blank value $\perp$. We use z to range both over values and variables.

A *page* is either the constant error or a map f representing the DOM of the page. The error page denotes that an error has occurred while processing a request at the server-side. The map f associates tags (i.e., strings) to links and HTML forms contained in the page. We represent them using the notation $\mathsf{form}(u, \vec{z})$, where u is the target URL and $\vec{z}$ is the list of parameters provided via the query string of a link or in the HTTP body of the request for forms.

Memories are maps from references to values. We use them in the server to hold the values of the variables during the execution, while in the browser they are used to model the cookie jar. We stipulate that $M(r) = \perp$ if $r \notin \mathit{dom}(M)$, i.e., the access to a reference not in memory yields a blank value.

Server Model

we let se range over expressions including variables, references, values, sampling of a fresh name (with label ℓ), e.g., to generate fresh cookie values, and binary operations. Server-side applications are represented as commands featuring standard programming constructs and special instructions for session establishment and management. Command $\mathbf{login}\ se_u, se_{pw}, se_{id}$ models a login operation with username se_u and password se_{pw}. The identity of the user is bound to the session identifier obtained by evaluating se_{id}. Command $\mathbf{start}\ se$ starts a new session or restores a previous one identified by the value of the expression se. Command $\mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell$ produces an authenticated event that includes data identified by the list of expressions $\vec{se}$. The command is annotated with a label ℓ denoting the expected security level of the event which has a central role in the security definition presented in subsection 5.3.5. Commands $\mathbf{if}\ \mathbf{tokenchk}(x, r)\ \mathbf{then}\ c$ and $\mathbf{if}\ \mathbf{originchk}(L)\ \mathbf{then}\ c$ respectively model a token check, comparing the value of a parameter x against the value of the reference r, and an origin check, verifying whether the origin of the request occurs in the set L. These checks are used as a protection mechanism against CSRF attacks. Command $\mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se}$ outputs an HTTP response containing a *page*, a script s and a sequence of `Set-Cookie` headers represented by the map ck. This command is a binder for $\vec{x}$ with scope $page, s, ck$, that is, the occurrences of the variables $\vec{x}$ in

$page, s, ck$ are substituted with the values obtained by evaluating the corresponding expressions in $\vec{se}$. Command $\mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x}$ outputs a message redirect to URL u with parameters $\vec{z}$ that sets the cookies in ck. This command is a binder for $\vec{x}$ with scope $\vec{z}, ck$.

Server code is evaluated using two memories: a global memory, freshly allocated when a connection is received, and a session memory, that is preserved across different requests. We write $@r$ and $\$r$ to denote the reference r in the global memory and in the session memory respectively. To link an executing command to its memories, we use an environment, which is a pair whose components identify the global memory and the session memory ($\perp$ when there is no active session).

The state of a server is modeled as a triple (D, ϕ, t) where the database D is a partial map from names to memories, ϕ maps session identifiers (i.e., names) to the corresponding user identities, and t is the parallel composition of multiple threads. Thread $u[\vec{r}](\vec{x}) \hookrightarrow c$ waits for an incoming connection to URL u and runs the command c when it is received. Lists $\vec{r}$ and $\vec{x}$ are respectively the list of cookies and parameters that the server expects to receive from the browser. Thread $\lceil c \rfloor_E^R$ denotes the execution of the command c in the environment E which identifies the memories of D on which the command operates. R tracks information about the request that triggered the execution, including the identifier n of the connection where the response by the server must be sent back, the URL of the endpoint u, the user ι who sent the request, and the origin of the request l. The user identity has no semantic import, but it is needed to spell out our security property.

User Behavior

action halt is used when an unexpected error occurs while browsing to prevent the user from performing further actions. Action $\mathsf{load}(tab, u, p)$ models the user entering the URL u in the address bar of her browser in tab, where p are the provided query parameters. Action $\mathsf{submit}(tab, u, v, p)$ models the user submitting a form or clicking on a link (identified by v) contained in the page at u rendered in tab; the parameters p are the inputs provided by the user. We represent user inputs as maps from integers to values v^τ annotated with their security type τ. In other words, we model that the user is aware of the security import of the provided parameters, e.g., whether a certain input is a password that must be kept confidential or a public value.

Browser Model

due to space constraints, we present the browser model in Appendix B.1.1. In the following we write $B_\iota(M, P, \vec{a})$ to represent a browser without any active script or open network connection, with cookie jar M and open pages P which is run by the user ι performing the list of actions $\vec{a}$.

Web Systems

the state of a web system is the parallel composition of the states of browsers and servers in the system. The state of an attacked web system also includes the attacker, modeled as a pair $(\ell, \mathcal{K})$ where the label ℓ defines the attacker power and $\mathcal{K}$ is her knowledge, i.e., a set of names that the attacker has learned by exploiting her capabilities.

5.3.3 Labels and Threat Model

Let $d \in \mathcal{D}$ be a domain and $\sim$ be the equivalence relation inducing the partition of $\mathcal{D}$ in sets of related domains.[2] We define the set of simple labels $\mathcal{L}$, ranged over by l, as the smallest set generated by the grammar:

$$l ::= \mathsf{http}(d) \mid \mathsf{https}(d) \mid l \vee l \mid l \wedge l$$

Intuitively, simple labels represent the entities entitled to read or write a certain piece of data, inspect or modify the messages exchanged over a network connection and characterize the capabilities of an attacker. A label ℓ is a pair of simple labels (l_C, l_I), where l_C and l_I are respectively the confidentiality and integrity components of ℓ. We let $C(\ell) = l_C$ and $I(\ell) = l_I$. We define the confidentiality pre-order $\sqsubseteq_C$ as the smallest pre-order on $\mathcal{L}$ closed under the following rules:

$$\frac{i \in \{1,2\}}{l_i \sqsubseteq_C l_1 \vee l_2} \qquad \frac{i \in \{1,2\}}{l_1 \wedge l_2 \sqsubseteq_C l_i}$$

$$\frac{l_1 \sqsubseteq_C l_3 \qquad l_2 \sqsubseteq_C l_3}{l_1 \vee l_2 \sqsubseteq_C l_3} \qquad \frac{l_1 \sqsubseteq_C l_2 \qquad l_1 \sqsubseteq_C l_3}{l_1 \sqsubseteq_C l_2 \wedge l_3}$$

We define the integrity pre-order $\sqsubseteq_I$ on simple labels such that $\forall l, l' \in \mathcal{L}$ we have $l \sqsubseteq_I l'$ iff $l' \sqsubseteq_C l$, i.e., confidentiality and integrity are contra-variant. For $\sqsubseteq_C$ we define the operators $\sqcup_C$ and $\sqcap_C$ that respectively take the least upper bound and the greatest lower bound of two simple labels. We define analogous operators $\sqcup_I$ and $\sqcap_I$ for $\sqsubseteq_I$. We let $\ell \sqsubseteq \ell'$ iff $C(\ell) \sqsubseteq_C C(\ell') \wedge I(\ell) \sqsubseteq_I I(\ell')$. We also define bottom and top elements of the lattices as follows:

$$\begin{aligned} \bot_C &= \textstyle\bigwedge_{d \in \mathcal{D}} (\mathsf{http}(d) \wedge \mathsf{https}(d)) & \bot_I &= \top_C \\ \top_C &= \textstyle\bigvee_{d \in \mathcal{D}} (\mathsf{http}(d) \vee \mathsf{https}(d)) & \top_I &= \bot_C \\ \bot &= (\bot_C, \bot_I) & \top &= (\top_C, \top_I) \end{aligned}$$

We label URLs, user actions and cookies by means of the function λ. We label URLs with their origin, i.e., given $u = (\pi, d, v)$ we let $\lambda(u) = (\pi(d), \pi(d))$. The label is used to: 1. characterize the capabilities required by an attacker to read and modify the contents of messages exchanged over network connections towards u; 2. identify which cookies are sent to and can be set by u. The label of an action is the one of its URL, i.e., we let $\lambda(a) = \lambda(u)$ for $a = \mathsf{load}(tab, u, p)$ and $a = \mathsf{submit}(tab, u, v, p)$.

The labelling of cookies depends on several aspects, e.g., the attributes specified by the web developer. For instance, a cookie for the domain d is given the following label:

$$(\mathsf{http}(d) \wedge \mathsf{https}(d), \textstyle\bigwedge_{d' \sim d} (\mathsf{http}(d') \wedge \mathsf{https}(d')))$$

[2] Two domains are related if they share the same base domain, i.e., the first upper-level domain which is not included in the public suffix list [ZJL+15]. For instance, `www.example.com` and `atk.example.com` are related domains, while `example.co.uk` and `atk.co.uk` are not.

The confidentiality label models that the cookie can be sent to d both over cleartext and encrypted connections, while the integrity component says that the cookie can be set by any of the related domains of d over any protocol, as dictated by the lax variant of the *Same Origin Policy* applied to cookies.

When the `Secure` attribute is used, the cookie is attached exclusively to HTTPS requests. However, `Secure` cookies can be set over HTTP [Bar11], hence the integrity is unchanged.[3] This behavior is represented by the following label:

$$(\mathsf{https}(d), \bigwedge_{d' \sim d}(\mathsf{http}(d') \wedge \mathsf{https}(d')))$$

Cookie prefixes [Wes] are a novel proposal aimed at providing strong integrity guarantees for certain classes of cookies. In particular, compliant browsers ensure that cookies having names starting with the `__Secure-` prefix are set over HTTPS and the `Secure` attribute is set. In our label model they can be represented as follows:

$$(\mathsf{https}(d), \bigwedge_{d' \sim d} \mathsf{https}(d'))$$

The `__Host-` prefix strengthens the policy enforced by `__Secure-` by additionally requiring that the `Domain` attribute is not set, thus preventing related domains from setting it. This is modeled by assigning the cookie the following label:

$$(\mathsf{https}(d), \mathsf{https}(d))$$

We discuss now the impact of HSTS [HJB12] on cookie labels. We use a set of domains $\Delta \subseteq \mathcal{D}$ to represent all the domains where HSTS is enabled, which essentially corresponds to the HSTS preload list[4] that is shipped with modern browsers. Since HSTS prevents browsers from communicating with certain domains over HTTP, in practice it prevents network attackers from setting cookies by modifying HTTP responses coming from these domains. The label of a `Secure` cookie for domain d becomes the following:

$$(\mathsf{https}(d), \bigwedge_{\substack{d' \sim d \\ d' \notin \Delta}} \mathsf{http}(d') \wedge \bigwedge_{d' \sim d} \mathsf{https}(d')))$$

The integrity label shows that the cookie can be set over HTTPS by any related domain of d (as for `Secure` cookies) and over HTTP only by related domains where HSTS is not enabled. If HSTS is activated for d and all its related domains, the cookie label becomes the same as that of cookies with the `__Secure-` prefix.

In the model we can also formalize attackers using labels which denote their read and write capabilities. Considering an attacker at label ℓ_a and a name with label ℓ, the name may be learned by the attacker if $C(\ell) \sqsubseteq_C C(\ell_a)$ and may be influenced by the attacker if $I(\ell_a) \sqsubseteq_I I(\ell)$. Here we model the following popular attackers from the web security literature:

[3] Although most modern browsers forbid this dangerous practice, we have decided to represent the behavior dictated by the cookie specification.

[4] `https://hstspreload.org`

1. The web attacker hosts a malicious website on domain d. We assume that the attacker owns a valid certificate for d, thus the website is available both over HTTP and HTTPS:

$$(\mathsf{http}(d) \vee \mathsf{https}(d), \mathsf{http}(d) \vee \mathsf{https}(d))$$

2. The active network attacker can read and modify the contents of all HTTP communications:

$$(\bigvee_{d \in \mathcal{D}} \mathsf{http}(d), \bigvee_{d \in \mathcal{D}} \mathsf{http}(d))$$

3. The related-domain attacker is a web attacker who hosts her website on a *related domain* of a domain d, thus she can set (domain) cookies for d. Assuming (for simplicity) that the attacker controls all the related domains of d, we can represent her capabilities with the following label:

$$(\bigvee_{\substack{d' \sim d \\ d' \neq d}} (\mathsf{http}(d') \vee \mathsf{https}(d')), \\ \bigvee_{\substack{d' \sim d \\ d' \neq d}} (\mathsf{http}(d') \vee \mathsf{https}(d')))$$

5.3.4 Semantics

We present now the most relevant rules of semantics in Table 5.2, deferring to Appendix B.1.3 for a complete formalization. In the rules we use the ternary operator "?:" with the usual meaning: $e \ ? \ e' : e''$ evaluates to e' if e is true, to e'' otherwise.

Servers

rules rely on the function $eval_E(se, D)$ that evaluates the expression se in the environment E using the database D. The formal definition is in Appendix B.1.3, here we provide an intuitive explanation. The evaluation of $@r$ and $\$r$ yields the value associated to r in the global and the session memory identified by E, respectively. Expression $\mathit{fresh}()^\ell$ evaluates to a fresh name sampled from $\mathcal{N}$ with security label ℓ. A value evaluates to itself. Evaluation of binary operations is standard.

Rule (S-RECV) models the receiving of a connection n at the endpoint u, as indicated by the action $\mathsf{req}(\iota_b, n, u, p, ck, l)$. A new thread is spawned where command c is executed after substituting all the occurrences of variables in $\vec{x}$ with the parameters p received from the network. We use the value $\perp$ for uninitialized parameters. The environment is $i, \perp$ where i identifies a freshly allocated global memory and $\perp$ that there is no ongoing session. The references of the global memory in $\vec{r}$ are initialized with the values in ck (if provided). In the request context we include the details about the incoming connection, including the origin l of the page that produced the request (or $\perp$, e.g., when the user opens the page in a new tab). The thread keeps listening for other connections on the same endpoint.

The evaluation of command **start** se is modeled by rules (S-RESTORESESSION) and (S-NEWSESSION). If se evaluates to a name $j \in \mathit{dom}(D)$, we resume a previously established session, otherwise we create a new one and allocate a new empty memory that is added to the

Servers

(S-RECV)
$$\frac{\alpha = \mathsf{req}(\iota_b, n, u, p, ck, l) \qquad R = n, u, \iota_b, l \qquad i \leftarrow \mathcal{N} \qquad \forall k \in [1 \ldots |\vec{r}|].\, M(r_k) = (r_k \in \mathit{dom}(ck))\ ?\ ck(r_k) : \bot \qquad m = |\vec{x}| \qquad \forall k \in [1 \ldots m].\, v_k = (k \in \mathit{dom}(p))\ ?\ p(k) : \bot \qquad \sigma = [x_1 \mapsto v_1, \ldots, x_m \mapsto v_m]}{(D, \phi, u[\vec{r}](\vec{x}) \hookrightarrow c) \xrightarrow{\alpha} (D \uplus \{i \mapsto M\}, \phi, \lceil c\sigma \rfloor^R_{i,\bot} \parallel u[\vec{r}](\vec{x}) \hookrightarrow c)}$$

(S-RESTORESESSION)
$$\frac{E = i, \_ \qquad \mathit{eval}_E(se, D) = j \qquad j \in \mathit{dom}(D)}{(D, \phi, \lceil \mathbf{start}\ se \rfloor^R_E) \xrightarrow{\bullet} (D, \phi, \lceil \mathbf{skip} \rfloor^R_{i,j})}$$

(S-NEWSESSION)
$$\frac{E = i, \_ \qquad \mathit{eval}_E(se, D) = j \qquad j \notin \mathit{dom}(D)}{(D, \phi, \lceil \mathbf{start}\ se \rfloor^R_E) \xrightarrow{\bullet} (D \uplus \{j \mapsto \{\}\}, \phi, \lceil \mathbf{skip} \rfloor^R_{i,j})}$$

(S-LOGIN)
$$\frac{R = n, u, \iota_b, l \qquad \mathit{eval}_E(se_u, D) = \iota_s \qquad \mathit{eval}_E(se_{pw}, D) = \rho(\iota_s, u) \qquad \mathit{eval}_E(se_{id}, D) = j}{(D, \phi, \lceil \mathbf{login}\ se_u, se_{pw}, se_{id} \rfloor^R_E) \xrightarrow{\bullet} (D, \phi \triangleleft \{j \mapsto \iota_s\}, \lceil \mathbf{skip} \rfloor^R_E)}$$

(S-OCHKSUCC)
$$\frac{R = n, u, \iota_b, l \qquad l \in L}{(D, \phi, \lceil \mathbf{if}\ \mathbf{originchk}(L)\ \mathbf{then}\ c \rfloor^R_E) \xrightarrow{\bullet} (D, \phi, \lceil c \rfloor^R_E)}$$

(S-TCHKFAIL)
$$\frac{\mathit{eval}_E(e_1, D) \neq \mathit{eval}_E(e_2, D)}{(D, \phi, \lceil \mathbf{if}\ \mathbf{tokenchk}(e_1, e_2)\ \mathbf{then}\ c \rfloor^R_E) \xrightarrow{\mathsf{error}} (D, \phi, \lceil \mathbf{reply}\ (\mathsf{error}, \mathbf{skip}, \{\}) \rfloor^R_E)}$$

(S-AUTH)
$$\frac{R = n, u, \iota_b, l \qquad j \in \mathit{dom}(\phi) \qquad \alpha = \sharp[\vec{v}]^{\iota_b, \phi(j)}_{\ell} \qquad \forall k \in [1 \ldots |\vec{se}|].\, \mathit{eval}_{i,j}(se_k, D) = v_k}{(D, \phi, \lceil \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell \rfloor^R_{i,j}) \xrightarrow{\alpha} (D, \phi, \lceil \mathbf{skip} \rfloor^R_{i,j})}$$

(S-REPLY)
$$\frac{R = n, u, \iota_b, l \qquad m = |\vec{x}| = |\vec{se}| \qquad \forall k \in [1, m].\, \mathit{eval}_E(se_k, D) = v_k \qquad \sigma = [x_1 \mapsto v_1, \ldots, x_m \mapsto v_m] \qquad \alpha = \overline{\mathsf{res}}(n, u, \bot, \langle\rangle, ck\sigma, \mathit{page}\sigma, s\sigma)}{(D, \phi, \lceil \mathbf{reply}\ (\mathit{page}, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se} \rfloor^R_E) \xrightarrow{\alpha} (D, \phi, \lceil \mathbf{halt} \rfloor^R_E)}$$

Table 5.2: Semantics (excerpt).

Web systems

$$\text{(A-BROSER)}\quad \frac{W \xrightarrow{\overline{\mathsf{req}}(\iota_b,n,u,p,ck,l)} W' \qquad W' \xrightarrow{\mathsf{req}(\iota_b,n,u,p,ck,l)} W'' \qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell))\ ?\ (\mathcal{K} \cup ns(p,ck)) : \mathcal{K}}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\bullet} (\ell,\mathcal{K}') \triangleright W''}$$

$$\text{(A-BROATK)}\quad \frac{\alpha = \overline{\mathsf{req}}(\iota_b,n,u,p,ck,l) \qquad W \xrightarrow{\alpha} W' \qquad I(\ell) \sqsubseteq_I I(\lambda(u)) \qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell))\ ?\ (\mathcal{K} \cup ns(p,ck)) : \mathcal{K}}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell,\mathcal{K}' \cup \{n\}) \triangleright W'}$$

$$\text{(A-ATKSER)}\quad \frac{n \leftarrow \mathcal{N} \qquad \iota_b \neq \mathsf{usr} \qquad ns(p,ck) \subseteq \mathcal{K} \qquad \alpha = \mathsf{req}(\iota_b,n,u,p,ck,l) \qquad W \xrightarrow{\alpha} W'}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell,\mathcal{K} \cup \{n\}) \triangleright W'}$$

Table 5.2: Semantics (excerpt, continued)

database D. We write $E = i, \_$ to denote that the second component of E is immaterial. In both cases the environment is updated accordingly.

Rule (S-LOGIN) models a successful login attempt. For this purpose, we presuppose the existence of a global partial function ρ mapping the pair (ι_s, u) to the correct password where ι_s is the identity of the user and u is the login endpoint. The rule updates the trust mapping ϕ by associating the session identifier specified in the **login** command with the identity ι_s.

Rules (S-OCHKSUCC) and (S-TCHKFAIL) treat a successful origin check and a failed token check, respectively. In the origin check we verify that the origin of the request is in a set of whitelisted origins, while in the token check we verify that two tokens match. In case of success we execute the continuation, otherwise we respond with an error message. In case of a failure we produce the event error.

Rule (S-AUTH) produces the authenticated event $\sharp[\vec{v}]_\ell^{\iota_b,\iota_s}$ where $\vec{v}$ is data identifying the event, e.g., *paper* and *action* in the HotCRP example of subsection 5.2.2. The event is annotated with the identities ι_b, ι_s, representing the user running the browser and the account where the event occurred, and the label ℓ denoting the security level associated to the event.

Rule (S-REPLY) models a reply from the server over the open connection n as indicated by the action $\overline{\mathsf{res}}$. The response contains a page *page*, script s and a map of cookies ck, where all occurrences of variables in $\vec{x}$ are replaced with the evaluation results of the expressions in $\vec{se}$. The third and the fourth component of $\overline{\mathsf{res}}$ are the redirect URL and the corresponding parameters, hence we use $\perp$ to denote that no redirect happens. We stipulate that the execution terminates after performing the reply as denoted by the instruction **halt**.

Web Systems

the semantics of web systems regulates the communications among browsers, servers and the attacker. Rule (A-BROSER) synchronizes a browser sending a request $\overline{\mathsf{req}}$ with the server willing to process it, as denoted by the matching action req. Here the attacker does not play an active

role (as denoted by action •) but she may update her knowledge with new secrets if she can read the contents of the request, modeled by the condition $C(\lambda(u)) \sqsubseteq_C C(\ell)$.

Rule (A-BROATK) uniformly models a communication from a browser to a server controlled by the attacker and an attacker that is actively intercepting network traffic sent by the browser. These cases are captured by the integrity check on the origin of the URL u. As in the previous rule, the attacker updates her knowledge if she can access the communication's contents. Additionally, she learns the network identifier needed to respond to the browser. In the trace of the system we expose the action intercepted/forged by the attacker. Rule (A-ATKSER) models an attacker opening a connection to an honest server. We require that the identity denoting the sender of the message belongs to the attacker and that the contents of the request can be produced by the attacker using her knowledge. Sequential application of the two rules lets us model a network attacker acting as a man-in-the-middle to modify the request sent by a browser to an honest server.

5.3.5 Security Definition

On a high level, our definition of session integrity requires that for each trace produced by the attacked web system, there exists a matching trace produced by the web system without the attacker, which in particular implies that authenticated actions cannot be modified or forged by the attacker. Before formalizing this property, we introduce the notion of trace.

Definition 1. *The system A generates the trace $\gamma = \alpha_1 \cdot \ldots \cdot \alpha_k$ iff the system can perform a sequence of steps $A \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_k} A'$ for some A' (also written as $A \xrightarrow{\gamma}{}^* A'$).*

Traces include attacker actions, authenticated events $\sharp[\vec{v}]_\ell^{\iota_b,\iota_s}$ and • denoting actions without visible effects or synchronizations not involving the attacker. Given a trace γ, we write $\gamma \downarrow (\iota, \ell)$ for the projection containing only the authentication events of the type $\sharp[\vec{v}]_\ell^{\iota_b,\iota_s}$ with $\iota \in \{\iota_b, \iota_s\}$. A trace γ is *unattacked* if it contains only • actions, error events and authenticated events, otherwise γ is an *attacked* trace.

Now we introduce the definition of session integrity.

Definition 2. *A web system W preserves* session integrity *against the attacker $(\ell_a, \mathcal{K})$ for the honest user* usr *performing the actions $\vec{a}$ if for any attacked trace γ generated by the system $(\ell_a, \mathcal{K}) \triangleright B_{\mathsf{usr}}(\{\}, \{\}, \vec{a}) \parallel W$ there exists an unattacked trace γ' generated by the same system such that for all labels ℓ we have:*

$$I(\ell_a) \not\sqsubseteq_I I(\ell) \Rightarrow \gamma \downarrow (\mathsf{usr}, \ell) = \gamma' \downarrow (\mathsf{usr}, \ell).$$

Intuitively, this means that the attacker can only produce authenticated events in her account or influence events produced by servers under her control. Apart from this, the attacker can only stop on-going sessions of the user but cannot intrude into them: this is captured by the existential quantification over unattacked traces that also lets us pick a prefix of any trace.

5.4 Security Type System

We now present a security type system designed for the verification of session integrity on web applications. It consists of several typing judgments covering server programs and browser scripts. Due to space constraints, in this Section we cover only the part related to server-side code and refer to Appendix B.1.4 for the typing rules of browser scripts.

5.4.1 Types

We introduce security types built upon the labels defined in subsection 5.3.3. We construct the set of security types $\mathcal{T}$, ranged over by τ, according to the following grammar:

$$\tau ::= \ell \mid \texttt{cred}(\ell)$$

We also introduce the set of reference types $\mathcal{T}_{\mathcal{R}} = \{\texttt{ref}(\tau) \mid \tau \in \mathcal{T}\}$ used for global and session references and we define the following projections on security types:

$$label(\ell) = \ell \quad label(\texttt{cred}(\ell)) = \ell$$
$$I(\tau) = I(label(\tau)) \quad C(\tau) = C(label(\tau))$$

Security types extend the standard security lattice with the type $\texttt{cred}(\ell)$ for credentials of label ℓ. We define the pre-order $\sqsubseteq_{\ell_a}$, parametrized by the attacker label ℓ_a, with the following rules:

$$\frac{\ell \sqsubseteq \ell'}{\ell \sqsubseteq_{\ell_a} \ell'} \qquad \frac{\begin{array}{c} C(\tau) \sqcup_C C(\tau') \sqsubseteq_C C(\ell_a) \\ I(\ell_a) \sqsubseteq_I I(\tau) \sqcap_I I(\tau') \end{array}}{\tau \sqsubseteq_{\ell_a} \tau'}$$

Intuitively, security types inherit the subtyping relation for labels but this is not lifted to the credentials, e.g., treating public values as secret credentials is unsound. However, types of low integrity and confidentiality (compared to the attacker's label) are always subtype of each other: in other words, we collapse all such types into a single one, as the attacker controls these values and is not limited by the restrictions enforced by types.

5.4.2 Typing Environment

Our typing environment $\Gamma = (\Gamma_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}})$ is a 5-tuple and conveys the following information:

- $\Gamma_{\mathcal{U}} : \mathcal{U} \to (\mathcal{L}^2 \times \vec{\mathcal{T}} \times \mathcal{L})$ maps URLs to labels capturing the security of the network connection, the types of the URL parameters and the integrity label of the reply;
- $\Gamma_{\mathcal{X}} : \mathcal{X} \to \mathcal{T}$ maps variables to types;
- $\Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$} : \mathcal{R} \to \mathcal{T}_{\mathcal{R}}$ map global references and session references, respectively, to reference types;

- $\Gamma_{\mathcal{V}} : \mathcal{V} \to (\mathcal{L}^2 \times \vec{\mathcal{T}} \times \mathcal{L})$ maps values used as tags for forms in the DOM to the corresponding type. We typically require the form's type to match the one of the form's target URL.

Now we introduce the notion of *well-formedness* which rules out inconsistent type assignments.

Definition 3. *A typing environment* Γ *is* well-formed *for* λ *and* ℓ_a *(written* $\lambda, \ell_a, \Gamma \vdash \diamond$*) if the following conditions hold:*

1. *for all URLs* $u \in \mathcal{U}$ *with* $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ *we have:*
   a) $C(\ell_u) = C(\lambda(u)) \wedge I(\lambda(u)) \sqsubseteq_I I(\ell_u)$
   b) *for all* $k \in [1 \ldots |\vec{\tau}|]$ *we have*
      i. $C(\tau_k) \sqsubseteq_C C(\ell_u) \wedge I(\ell_u) \sqsubseteq_I I(\tau_k)$
      ii. $\tau_k = \texttt{cred}(\cdot) \wedge C(\tau_k) \sqsubseteq_C C(\ell_a) \Rightarrow I(\ell_a) \sqsubseteq_I I(\tau_k)$
2. *for all references* $r \in \mathcal{R}$ *with* $\Gamma_{\mathcal{R}^@}(r) = \tau$*:*
   a) $C(\tau) \sqsubseteq_C C(\lambda(r)) \wedge I(\lambda(r)) \sqsubseteq_I I(\tau)$
   b) *for all* $u \in \mathcal{U}$*, if* $C(\lambda(r)) \sqsubseteq_C (\lambda(u)) \wedge I(\ell_a) \sqsubseteq_I I(\lambda(u))$ *then* $C(\tau) \sqsubseteq_C C(\ell_a)$
   c) *if* $I(\ell_a) \sqsubseteq_I I(\lambda(r))$ *and* $\tau = \texttt{cred}(\cdot)$ *then* $C(\tau) \sqsubseteq_C C(\ell_a)$
   d) $\tau = \texttt{cred}(\cdot) \wedge C(\tau) \sqsubseteq_C C(\ell_a) \Rightarrow I(\ell_a) \sqsubseteq_I I(\tau)$

Conditions (1a) and (2a) ensure that the labels of URLs and cookies in the typing environment – which are used for the security analysis – are at most as strict as the labels in the function λ introduced in subsection 5.3.3 – which define the semantics. For instance, a cookie r with confidentiality label $C(\lambda(r)) = \texttt{http}(d) \wedge \texttt{https}(d)$ is attached both to HTTP and HTTPS requests to domain d. It would be unsound to use a stronger label for typing, e.g., $\texttt{https}(d)$, since we would miss attacks due to the cookie leakage over HTTP. In the same spirit, we check that URLs do not contain parameters requiring stronger type guarantees than those offered by the type assigned to the URL (1b i).

Conditions (1b ii) and (2d) ensure that low confidentiality credentials – that can be learned and used by the attacker – cannot have high integrity.

Additionally, well-formedness rules out two inherently insecure type assignments for cookies. First, if a low integrity URL can read a cookie, then the cookie must have low confidentiality since the attacker can inject a script leaking the cookies, as in a typical XSS (2b). Second, cookies that can be set over a low integrity network connection cannot be high confidentiality credentials since the attacker can set them to a value she knows (2c). ci

5.4.3 Intuition Behind the Typing Rules

The type system resembles one for standard information flow control (IFC) where we consider explicit and implicit flows for integrity, but only explicit flows for confidentiality: since our property of interest is web session integrity, regarding confidentiality we are only interested in preventing credentials from being leaked (since they are used for access control), while the leakage of other values does not impact our property. The type system restricts the operations on credentials to be equality checks, hence the leak of information through implicit flows is limited to one bit: this is consistent with the way credentials are handled by real web applications. A treatment of implicit flows for confidentiality would require a declassification mechanism to handle the bit leaked by credential checks, thus complicating our formalism without adding any tangible security guarantee.

As anticipated in section 5.2, the code is type-checked twice under different assumptions: first, we consider the case of an honest user visiting the server; second, we consider a CSRF attempt where the attacker forces the user's browser to send a request to the server. We do not consider the case of the attacker visiting the server from her own browser since we can prove that such a session is always well-typed, which is close in spirit to the opponent typability lemma employed in type systems for cryptographic protocols [FM11, BHM14].

To enforce our session integrity property, the type system needs to track the identity of the user owning the session and the intention of the user to perform authenticated actions. In typing, this is captured by two dedicated labels.

The *session label* ℓ_s records the owner of the active session and is used to label references in the session memory. The label typically equals the one of the session identifier, thus it changes when we resume or start a new session. Formally, $\ell_s \in \mathcal{L}^2 \cup \{\times\}$ where $\times$ denotes no active session.

The *program counter label* $\mathtt{pc} \in \mathcal{L}$ tracks the integrity of the control flow. A high $\mathtt{pc}$ implies that the control flow is intended by the user. The $\mathtt{pc}$ is lowered in conditionals with a low integrity guard, as is standard in IFC type systems. In the CSRF typing branch, the $\mathtt{pc}$ will be permanently low: we need to prune this typing branch to type-check high integrity actions. For this purpose, we use token or origin checks: in the former, the user submits a CSRF token that is compared to a (secret) session reference or cookie, while in the latter we check whether the origin of the request is contained in a whitelist. There are cases in which we statically know that the check will fail, allowing us to prune typing branches.

We also briefly comment on another important attack, namely cross-site scripting (XSS): we can model XSS vulnerabilities by including a script from an attacker-controlled domain, which causes a failure in typing. However, XSS prevention is orthogonal to the goal of our work and must be solved with alternative techniques, e.g., proper input filtering or CSP [W3C16].

5.4.4 Explanation of the Typing Rules

Server Expressions

typing of server expressions is ruled by the judgement $\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau$, meaning that the expression se has type τ in the typing environment Γ within the session ℓ_s. Names have type

Server expressions

(T-ENAME)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} n^{\ell} : \mathtt{cred}(\ell)}$$

(T-EFRESH)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \mathit{fresh}()^{\ell} : \mathtt{cred}(\ell)}$$

(T-EVAL)
$$\frac{v \notin \mathcal{N}}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} v : \bot}$$

(T-EUNDEF)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \bot : \tau}$$

(T-EVAR)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} x : \Gamma_{\mathcal{X}}(x)}$$

(T-EGLOBREF)
$$\frac{\Gamma_{\mathcal{R}^{@}}(r) = \mathtt{ref}(\tau)}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} @r : \tau}$$

(T-ESESREF)
$$\frac{\begin{array}{c}\ell_s \neq \times \qquad \Gamma_{\mathcal{R}^{\$}}(r) = \mathtt{ref}(\tau') \\ \ell = (C(\tau') \sqcap_C C(\ell_s), I(\tau') \sqcup_I I(\ell_s)) \\ \tau = (\tau' \neq \mathtt{cred}(\cdot))\ ?\ \ell\ :\ \mathtt{cred}(\ell)\end{array}}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \$r : \tau}$$

(T-EBINOP)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se' : \tau' \\ (\tau, \tau' \neq \mathtt{cred}(\cdot)) \vee \odot \text{ is } =\end{array}}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se \odot se' : \mathit{label}(\tau) \sqcup \mathit{label}(\tau')}$$

(T-ESUB)
$$\frac{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau' \qquad \tau' \sqsubseteq_{\ell_a} \tau}{\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau}$$

Server references

(T-RGLOBREF)
$$\frac{}{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} @r : \Gamma_{\mathcal{R}^{@}}(r)}$$

(T-RSUB)
$$\frac{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau') \qquad \tau \sqsubseteq_{\ell_a} \tau'}{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau)}$$

(T-RSESREF)
$$\frac{\begin{array}{c}\ell_s \neq \times \qquad \Gamma_{\mathcal{R}^{\$}}(r) = \mathtt{ref}(\tau') \qquad \ell = (C(\tau') \sqcap_C C(\ell_s), I(\tau') \sqcup_I I(\ell_s)) \\ \tau = (\tau' \neq \mathtt{cred}(\cdot))\ ?\ \ell\ :\ \mathtt{cred}(\ell)\end{array}}{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} \$r : \mathtt{ref}(\tau)}$$

Table 5.3: Type system.

Server-side commands

(T-SKIP)
$$\frac{}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{skip} : \ell_s, \mathrm{pc}}$$

(T-SEQ)
$$\frac{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} c : \ell_s', \mathrm{pc}' \quad \Gamma, \ell_s', \mathrm{pc}' \vdash^{\mathsf{c}}_{\ell_a, C} c' : \ell_s'', \mathrm{pc}''}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} c; c' : \ell_s'', \mathrm{pc}''}$$

(T-IF)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \qquad \mathrm{pc}' = \mathrm{pc} \sqcup_I I(\tau) \\ \Gamma, \ell_s, \mathrm{pc}' \vdash^{\mathsf{c}}_{\ell_a, C} c : \ell_s'', \mathrm{pc}_1 \qquad \Gamma, \ell_s, \mathrm{pc}' \vdash^{\mathsf{c}}_{\ell_a, C} c' : \ell_s''', \mathrm{pc}_2 \\ \mathrm{pc}'' = (\mathbf{reply}, \mathbf{redirect} \in \mathit{coms}(c) \cup \mathit{coms}(c'))\ ?\ \mathrm{pc}_1 \sqcup_I \mathrm{pc}_2\ :\ \mathrm{pc} \\ \ell_s' = (\ell_s'' = \ell_s''')\ ?\ \ell_s''\ :\ \times\end{array}}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{if}\ se\ \mathbf{then}\ c\ \mathbf{else}\ c' : \ell_s', \mathrm{pc}''}$$

(T-LOGIN)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_u : \tau \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_{pw} : \mathtt{cred}(\ell) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_{sid} : \mathtt{cred}(\ell') \\ C(\mathtt{cred}(\ell)) \sqsubseteq_C C(\mathtt{cred}(\ell')) \qquad I(\tau) \sqcup_I I(\mathtt{cred}(\ell)) \sqcup_I \mathrm{pc} \sqsubseteq_I I(\mathtt{cred}(\ell'))\end{array}}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{login}\ se_u, se_{pw}, se_{sid} : \ell_s, \mathrm{pc}}$$

(T-START)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \mathtt{cred}(\ell) \\ \ell_s' = (C(\mathtt{cred}(\ell)) \sqsubseteq_C C(\ell_a))\ ?\ (\bot_C, \top_I) : \ell \\ b = \mathsf{hon} \Rightarrow ((\ell_s = \times \vee \mathrm{pc} \sqsubseteq_I I(\ell_s)) \wedge \mathrm{pc} \sqsubseteq_I I(\ell_s'))\end{array}}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{start}\ se : \ell_s', \mathrm{pc}}$$

(T-SETGLOBAL)
$$\frac{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} @r : \mathtt{ref}(\tau) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \qquad \mathrm{pc} \sqsubseteq_I I(\tau)}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} @r := se : \ell_s, \mathrm{pc}}$$

(T-SETSESSION)
$$\frac{\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} \$r : \mathtt{ref}(\tau) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se : \tau \qquad \mathrm{pc} \sqsubseteq_I I(\tau)}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \$r := se : \ell_s, \mathrm{pc}}$$

(T-PRUNETCHK)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\mathtt{cred}(\ell)) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} x : \tau \\ C(\tau) \neq C(\mathtt{cred}(\ell)) \qquad C(\mathtt{cred}(\ell)) \not\sqsubseteq_C C(\ell_a) \qquad b = \mathsf{csrf}\end{array}}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, (u, b, \mathcal{P})} \mathbf{if}\ \mathbf{tokenchk}(x, r)\ \mathbf{then}\ c : \ell_s, \mathrm{pc}}$$

(T-TCHK)
$$\frac{\begin{array}{c}\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\mathtt{cred}(\ell)) \qquad \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} x : \mathtt{cred}(\ell) \\ \Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} c : \ell_s', \mathrm{pc}\end{array}}{\Gamma, \ell_s, \mathrm{pc} \vdash^{\mathsf{c}}_{\ell_a, C} \mathbf{if}\ \mathbf{tokenchk}(x, r)\ \mathbf{then}\ c : \ell_s', \mathrm{pc}}$$

Table 5.3: Type system (continued).

Server-side commands (continued)

(T-PRUNEOCHK)

$$\frac{\forall l \in L. I(\ell_a) \not\sqsubseteq_I l \qquad u \in \mathcal{P} \qquad b = \mathsf{csrf}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \textbf{if originchk}(L) \textbf{ then } c : \ell_s, \mathtt{pc}}$$

(T-OCHK)

$$\frac{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,C} c : \ell_s', \mathtt{pc}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,C} \textbf{if originchk}(L) \textbf{ then } c : \ell_s', \mathtt{pc}}$$

(T-AUTH)

$$\frac{\begin{array}{c}\ell_s \neq \times \qquad \forall k \in [1 \ldots |\vec{se}|]. \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_k : \tau_k \\ \left(I(\ell_a) \sqsubseteq_I \bigsqcup_{I\ 1 \leq k \leq |\vec{se}|} I(\tau_k) \sqcup_I \mathtt{pc} \sqcup_I I(\ell_s)\right) \Rightarrow I(\ell_a) \sqsubseteq_I I(\ell)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,C} \textbf{auth } \vec{se} \textbf{ at } \ell : \ell_s, \mathtt{pc}}$$

(T-REPLY)

$$\frac{\begin{array}{c}\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad \mathtt{pc}' = \mathtt{pc} \sqcup_I l_r \qquad \Gamma'_{\mathcal{X}} = x_1 \colon \tau_1, \ldots, x_{|\vec{se}|} \colon \tau_{|\vec{se}|} \\ \Gamma' = (\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}}) \qquad \forall k \in [1 \ldots |\vec{se}|]. \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_k : \tau_k \wedge C(\tau_k) \sqsubseteq_C C(\ell_u) \\ \forall r \in \mathit{dom}(ck). \Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau_r) \wedge \Gamma', \ell_s \vdash^{\mathsf{se}}_{\ell_a} ck(r) : \tau_r \wedge \mathtt{pc}' \sqsubseteq_I I(\tau_r) \\ \Gamma', b, \mathtt{pc}' \vdash^{\mathsf{s}}_{\ell_a,\mathcal{P}} s \qquad b = \mathsf{csrf} \Rightarrow \forall x \in \mathit{vars}(s). C(\Gamma'_{\mathcal{X}}(x)) \sqsubseteq_C C(\ell_a) \\ b = \mathsf{hon} \Rightarrow \mathtt{pc} \sqsubseteq_I l_r \wedge \left(\mathit{page} = \mathsf{error} \vee \forall v \in \mathit{dom}(\mathit{page}). \Gamma', v, \mathtt{pc}' \vdash^{\mathsf{f}}_{\ell_a} \mathit{page}(v)\right) \\ I(\ell_a) \sqsubseteq_I I(\ell_u) \Rightarrow \forall k \in [1 \ldots |\vec{se}|]. C(\tau_k) \sqsubseteq_C C(\ell_a)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \textbf{reply } (\mathit{page}, s, ck) \textbf{ with } \vec{x} = \vec{se} : \ell_s, \mathtt{pc}}$$

(T-REDIR)

$$\frac{\begin{array}{c}\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad \Gamma'_{\mathcal{X}} = x_1 \colon \tau_1, \ldots, x_{|\vec{se}|} \colon \tau_{|\vec{se}|} \\ \Gamma' = (\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}}) \qquad \forall k \in [1 \ldots |\vec{se}|]. \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} se_k : \tau_k \wedge C(\tau_k) \sqsubseteq_C C(\ell_u) \\ \forall r \in \mathit{dom}(ck). \Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau_r) \wedge \Gamma', \ell_s \vdash^{\mathsf{se}}_{\ell_a} ck(r) : \tau_r \wedge \mathtt{pc} \sqsubseteq_I I(\tau_r) \\ I(\ell_a) \sqsubseteq_I I(\ell_u) \Rightarrow \forall k \in [1 \ldots |\vec{se}|]. C(\tau_k) \sqsubseteq_C C(\ell_a) \\ b = \mathsf{csrf} \Rightarrow \forall x \in \mathit{vars}(\vec{z}). C(\Gamma'_{\mathcal{X}}(x)) \sqsubseteq_C C(\ell_a) \\ u' \notin \mathcal{P} \qquad \Gamma_{\mathcal{U}}(u') = \ell'_u, \vec{\tau'}, l'_r \qquad l_r = l'_r \qquad I(\ell_a) \not\sqsubseteq_I I(\ell_u) \\ b = \mathsf{hon} \Rightarrow \left(\mathtt{pc} \sqsubseteq_I I(\ell_u) \wedge m = |\vec{z}| = |\vec{\tau'}| \wedge \forall k \in [1 \ldots m]. \Gamma', \ell_s \vdash^{\mathsf{se}}_{\ell_a} z_k : \tau'_k \wedge \tau'_k \sqsubseteq_{\ell_a} \tau_k\right)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \textbf{redirect } (u', \vec{z}, ck) \textbf{ with } \vec{x} = \vec{se} : \ell_s, \mathtt{pc}}$$

Table 5.3: Type system (continued).

Forms

(T-FORM)

$$\frac{\begin{array}{c}\Gamma_{\mathcal{V}}(v) = \Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad I(\ell_a) \not\sqsubseteq_I I(\ell_u) \\ \mathtt{pc} \sqsubseteq_I I(\ell_u) \qquad m = |\vec{z}| = |\vec{\tau}| \qquad \forall k \in [1 \ldots m].\, \Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} z_k : \tau'_k \wedge \tau'_k \sqsubseteq_{\ell_a} \tau_k\end{array}}{\Gamma, v, \mathtt{pc} \vdash^{\mathsf{f}}_{\ell_a} \mathsf{form}(u, \vec{z})}$$

Server threads

(T-PARALLEL)

$$\frac{\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t \qquad \Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t'}{\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t \parallel t'}$$

(T-RECV)

$$\frac{\begin{array}{c}\lambda, \ell_a, \Gamma^0 \vdash \diamond \qquad \Gamma^0_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad m = |\vec{\tau}| = |\vec{x}| \\ \forall k \in [1 \ldots |\vec{r}|].\, C(\Gamma^0_{\mathcal{R}^@}(r_k)) \sqsubseteq_C C(\ell_u) \wedge I(\ell_u) \sqsubseteq_I I(\Gamma^0_{\mathcal{R}^@}(r_k)) \\ \Gamma_{\mathcal{X}} = x_1 \colon \tau_1, \ldots, x_m \colon \tau_m \qquad (\Gamma^0_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma^0_{\mathcal{R}^@}, \Gamma^0_{\mathcal{R}^\$}, \Gamma^0_{\mathcal{V}}), \times, I(\ell_u) \vdash^{\mathsf{c}}_{\ell_a,(u,\mathsf{hon},\mathcal{P})} c : \_, I(\ell_u) \\ \Gamma'_{\mathcal{X}} = x_1 \colon (\bot_C, \top_I), \ldots, x_m \colon (\bot_C, \top_I) \\ (\Gamma^0_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma^0_{\mathcal{R}^@}, \Gamma^0_{\mathcal{R}^\$}, \Gamma^0_{\mathcal{V}}), \times, \top_I \vdash^{\mathsf{c}}_{\ell_a,(u,\mathsf{csrf},\mathcal{P})} c : \_, \top_I\end{array}}{\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} u[\vec{r}](\vec{x}) \hookrightarrow c}$$

Table 5.3: Type system (continued).

$\mathtt{cred}(\ell)$ where ℓ is the label provided as an annotation (T-ENAME, T-EFRESH). Values different from names are constants of type $\bot$, i.e., they have low confidentiality and high integrity (T-EVAL). Rule (T-EUNDEF) gives any type to the undefined value $\bot$. This is needed since the initial memory and empty parameters contain this value and have to be well-typed. Types for variables and references in the global memory are read from the corresponding environments (T-EVAR,T-EGLOBREF). For session references we combine the information stored in the environment with the session label ℓ_s, which essentially acts as an upper bound on the types of references (T-ESESREF). In a honest session, ℓ_s can have high confidentiality, thus the session memory can be used to store secrets. In the attacker session, instead, the types of all session references are lowered and can never store secrets. Typing fails if no session is active, i.e., $\ell_s = \times$. The computed type for a reference is a credential type if and only if it is so in the environment. Binary operations are given the join of the labels of the two operands (T-EBINOP). However, on credentials we allow only equality checks to limit leaks through implicit flows. Note that by projecting the types to their labels we perform a declassification and hence the result of a binary operation can never be a high confidentiality credential. Finally, (T-ESUB) lets us use subtyping on expressions.

Server References

typing of server references is ruled by the judgment $\Gamma, \ell_s \vdash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau)$ meaning that the reference r has type $\mathtt{ref}(\tau)$ in the typing environment Γ within the session ℓ_s. This judgement is used to derive the type of a reference we write into, in contrast to the typing of expressions which covers the typing of references from which we read. While (T-RGLOBREF) just looks up the type of the global reference in the typing environment, in (T-RSESREF) we have analogous conditions to (T-ESESREF) for session references. Subtyping for reference types is contra-variant to subtyping for security types (T-RSUB).

Server Commands

the judgement $\Gamma, \ell_s, \mathtt{pc} \vdash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell_s', \mathtt{pc}'$ states that the command c (bound to the endpoint at URL u) can be typed against the attacker ℓ_a in the typing branch $b \in \{\mathsf{hon}, \mathsf{csrf}\}$ using typing environment Γ, session label ℓ_s and program counter label $\mathtt{pc}$. $\mathcal{P}$ contains all URLs that rely on an origin check to prevent CSRF attacks. After the execution of c, the session label and the PC label are respectively updated to ℓ_s' and $\mathtt{pc}'$. We let $C = (u, b, \mathcal{P})$ if the individual components of the tuple are not used in a rule. The branch b tracks whether we are typing the scenario of an honest request ($b = \mathsf{hon}$) or the CSRF case ($b = \mathsf{csrf}$).

Rule (T-SKIP) does nothing, while (T-SEQ) types the second command with the session label and the PC label obtained by typing the first command.

Rule (T-LOGIN) verifies that the password and the session identifier are both credentials and that the latter is at least as confidential as the former, since the identifier can be used for authentication in place of the password. Finally, we check that the integrity of username, password and $\mathtt{pc}$ are at least as high as the integrity of the session identifier to prevent an unauthorized party from influencing the identity associated to the session.

Rule (T-START) updates the session label used for typing the following commands. First we check that the session identifier se has a credential type: if it has low confidentiality, we update the session label to $(\bot_C, \top_I)$ (since the attacker can access the session), otherwise we use the label ℓ in the type of se. Furthermore, we ensure that in the honest typing branch high integrity sessions can not be started or ended (by starting a new session) in a low integrity context (i.e., in a conditional with low integrity guard), since this can potentially influence the value of high integrity references of the session memory in the continuation. For the CSRF typing branch this is not required, since due to its low PC label it can never write to high integrity references.

Rules (T-SETGLOBAL) and (T-SETSESSION) ensure that no explicit flow violates the confidentiality or integrity policies, where for integrity we also consider the PC label.

Rule (T-IF) lowers the PC based on the integrity label of the guard expression of the conditional and uses it to type-check the two branches. If one of the branches contains a **reply** or a **redirect** command, then reaching the continuation depends on the taken branch, thus we use the join of the PC labels returned in the two branches to type-check the continuation; otherwise, we use the original PC label. If typing the two branches yields two different session labels, we use the

session label $\times$ in the continuation to signal that the session state cannot be statically predicted and thus no session operation should be allowed.

Rule (T-AUTH) ensures that the attacker cannot affect any component leading to an authenticated event (PC label, session label or any expression in $\vec{se}$) unless the event is annotated with a low integrity label. Since authenticated events are bound to sessions, we require $\ell_s \neq \times$.

Rules (T-PRUNETCHK) and (T-TCHK) handle CSRF token checks. In (T-PRUNETCHK) we statically know that the check fails since the reference where the token is stored has a high confidentiality credential type and the parameter providing the token is a low confidentiality value, hence we do not type-check the continuation c. This reasoning is sound since credentials are unguessable fresh names and we disallow subtyping for high confidentiality credentials, i.e., public values cannot be treated as secret credentials. This rule is used only in the CSRF typing branch. Rule (T-TCHK) covers the case where the check may succeed and we simply type-check the continuation c. We do not change the PC label since a failure in the check produces an error page which causes the user to stop browsing.

Similarly, rules (T-PRUNEOCHK) and (T-OCHK) cover origin checks. We can prune the CSRF typing branch if the URL we are typing is protected ($u \in \mathcal{P}$) and all whitelisted origins have high integrity, since the origin of a CSRF attack to a protected URL has always low integrity.

Rule (T-REPLY) combines the PC label with the expected integrity label of the response l_r for the current URL to compute $\mathtt{pc}'$ which is used to type the response. In the honest typing branch, we require $\mathtt{pc}' = l_r$, which establishes an invariant used when typing an **include** command in a browser script, where we require that the running script and the included script can be typed with the same $\mathtt{pc}$ (cf. rule (T-BINCLUDE) in Appendix B.1.4). Using the typing environment Γ' which contains types for the variables embedded in the response, we check the following properties:

- secrets are not disclosed over a network connection which cannot guarantee their confidentiality;
- the types of the values assigned to cookies are consistent with those in the typing environment (where the PC label is taken into account for the integrity component);
- the script in the response is well-typed (rules in Appendix B.1.4);
- secrets are not disclosed to a script in the CSRF typing branch since it might be included by an attacker's script;
- in the honest typing branch, we check that the returned page is either the error page or all its forms are well-typed according to rule (T-FORM). We do not perform this check in the CSRF branch since a CSRF attack is either triggered by a script inclusion or through a redirect. In the first case the attacker cannot access the DOM, which in a real browser is enforced by the Same Origin Policy. In the second case, well-formed user behavior (cf. Definition 4) ensures that the user will not interact with the DOM in this scenario;

- no high confidentiality data is included in replies over a low integrity network connection, since the attacker could inject scripts to leak secrets embedded in the response.

Rule (T-REDIR) performs mostly the same checks as (T-REPLY). Instead of typing script and DOM, we perform checks on the URL similar to the typing of forms, as discussed below. Additionally, we require that the target URL is not relying on an origin check for CSRF protection ($u' \notin \mathcal{P}$), as the redirect would allow for a circumvention of that protection. Finally, we also require that the expected integrity label for the response for the current URL and the target URL are the same.

Forms

the judgement $\Gamma, v, \mathtt{pc} \vdash^{\mathsf{f}}_{\ell_a} f$ says that a form f identified by the name v is well-typed in the environment Γ under the label $\mathtt{pc}$. Our rule for typing forms (T-FORM) first checks that the type of the form name matches the type of the target URL. This is needed since for well-formed user behavior (cf. Definition 4) we assume that the user relies on the name of a form to ensure that her inputs are compliant with the expected types. We require that only links to high integrity URLs are included and with $\mathtt{pc} \sqsubseteq_I I(\ell_u)$ we check that the thread running with program counter label $\mathtt{pc}$ is allowed to trigger requests to u. In this way we can carry over the $\mathtt{pc}$ from one thread where the form has been created to the one receiving the request since we type-check the honest branch with $\mathtt{pc} = I(\ell_u)$. Finally, we check that the types of form values comply with the expected type for the corresponding URL parameters, taking the PC into account for implicit integrity flows.

Server Threads

the judgement $\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a, \mathcal{P}} t$ says that the thread t is well-typed in the environment Γ^0 against the attacker ℓ_a and $\mathcal{P}$ is the set of URLs protected against CSRF attacks via origin checking.

Rule (T-PARALLEL) states that the parallel composition of two threads is well-typed if both are well-typed. Rules for typing running threads (i.e., $t = \lceil c \rfloor^R_E$) are in Appendix B.1.4, since they are needed only for proofs.

Rule (T-RECV) checks that the environment is well-formed and that the network connection type ℓ_u is strong enough to guarantee the types of the cookies, akin to what is done for parameters in Definition 3. Then we type-check the command twice with $\ell_s = \times$, since no session is initially active. In the first branch we let $b = \mathsf{hon}$: parameters are typed according to the type of u in $\Gamma^0_{\mathcal{U}}$ which is reflected in the environment $\Gamma_{\mathcal{X}}$. As the honest user initiated the request, we let $\mathtt{pc} = I(\ell_u)$, i.e., we use the integrity label of the network connection as $\mathtt{pc}$. This allows us to import information about the program counter from another (well-typed) server thread or browser script that injected the form into the DOM or directly triggered the request. In the second branch we let $b = \mathsf{csrf}$: parameters are chosen by the attacker, hence they have type $(\bot_C, \top_I)$ in $\Gamma'_{\mathcal{X}}$. As the attacker initiated the request, we let $\mathtt{pc} = \top_I$.

5.4.5 Formal Results

We introduce the notion of *navigation flow*, which identifies a sequence of navigations among different pages occurring in a certain tab and triggered by the user's interaction with the elements of the DOM of rendered pages. Essentially, a navigation flow is a list of user actions consisting of a load on a certain tab followed by all actions of type submit in that tab (modeling clicks on links and submissions of forms) up to the next load (if any). A formal definition is presented in Appendix B.1.5.

Next we introduce the notion of *well-formedness* to constrain the interactions of an honest user with a web system.

Definition 4. *The list of user actions $\vec{a}$ is* well-formed *for the honest user* usr *in a web system W with respect to a typing environment Γ^0 and an attacker ℓ_a iff*

1. *for all actions a' in $\vec{a}$ we have:*
   - *if $a' = \mathsf{load}(tab, u, p)$, $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ then for all $k \in dom(p)$ we have $p(k) = v^{\tau'} \Rightarrow \tau' \sqsubseteq_{\ell_a} \tau_k$;*
   - *if $a' = \mathsf{submit}(tab, u, v', p)$, $\Gamma_{\mathcal{V}}(v') = \ell_u, \vec{\tau}, l_r$ then for all $k \in dom(p)$ we have if $p(k) = v^{\tau'}$ then $\tau' \sqsubseteq_{\ell_a} \tau_k$. If $I(\ell_a) \sqsubseteq_I \lambda(u)$ we additionally have $\tau' \sqsubseteq_{\ell_a} \ell_a$.*
2. *$(\ell_a, \mathcal{K}_0) \triangleright B_{\mathsf{usr}}(\{\}, \{\}, \vec{a}) \parallel W \xrightarrow{\gamma}{}^{*} (\ell_a, \mathcal{K}') \triangleright B_{\mathsf{usr}}(M, P, \langle\rangle) \parallel W'$ for some $\mathcal{K}', W', M, P$ where γ is an unattacked trace, not containing the event* error*;*
3. *for every navigation flow $\vec{a}'$ in $\vec{a}$, we have that $I(\ell_a) \sqsubseteq_I I(\lambda(a'_j))$ implies $I(\ell_a) \sqsubseteq_I I(\lambda(a'_k))$ for all $j < k \leq |\vec{a'}|$.*

Condition 1 prevents the user from deliberately leaking secrets by enforcing that the expected parameter types are respected. While the URL in a load event is the target URL and we can directly check its type, in a submit action it refers to the page containing the form: intuitively, this models a user who knows which page she is actively visiting with a load and which page she is currently on when performing a submit. However, we do not expect the user to inspect the target URL of a form. Instead, we expect the user to identify a form by its displayed name (the parameter v' in submit) and input only data matching the type associated to that form name. For instance, in a form named "public comment", we require that the user enters only public data. Typing hence has to enforces that all forms the user interacts with are named correctly. Otherwise, an attacker could abuse a mismatch of form name and target URL in order to steal confidential data. For this reason we also require that the user never provides secrets to a form embedded in a page of low integrity.

Condition 2 lets us consider only honest runs in which the browser terminates regularly without producing errors. Concretely, this rules out interactions that deliberately trigger an error at the server-side, e.g., the user loads a page expecting a CSRF token without providing this token, or executions that do not terminate due to infinite loops, e.g., where a script recursively includes itself.

Condition 3 requires that the user does not navigate a trusted website reached by interacting with an untrusted page. Essentially, this rules out phishing attempts where the attacker influences the content shown to the user in the trusted website.

Our security theorem predicates over *fresh clusters*, i.e., systems composed of multiple servers where no command is running or has been run in the past.

Definition 5. *A server S is* fresh *if $S = (\{\}, \{\}, t)$ where t is the parallel composition of threads of the type $u[\vec{r}](\vec{x}) \hookrightarrow c$. A system W is a* fresh cluster *if it is the parallel composition of fresh servers.*

We now present the main technical result, namely that well-typed clusters preserve the session integrity property from Definition 2 for all well-formed interactions of the honest user with the system, provided that her passwords are confidential.

Theorem 1. *Let W be a fresh cluster, $(\ell_a, \mathcal{K})$ an attacker, Γ^0 a typing environment, $\mathcal{P}$ a set of protected URLs against CSRF via origin checking and let $\vec{a}$ be a list of well-formed user actions for* usr *in W with respect to Γ^0 and ℓ_a. Assume that for all u with $\rho(\mathsf{usr}, u) = n^\ell$ we have $C(\ell) \not\sqsubseteq_C C(\ell_a)$ and for all $n^\ell \in \mathcal{K}$ we have $C(\ell) \sqsubseteq_C C(\ell_a)$. Then W preserves* session integrity *against ℓ_a with knowledge $\mathcal{K}$ for the honest user* usr *performing the list of actions $\vec{a}$ if $\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$ for all servers $S = (\{\}, \{\}, t)$ in W.*

The proof builds upon a simulation relation connecting a run of the system with the attacker with a corresponding run of the system without the attacker in which the honest user behaves in the same way and high integrity authenticated events are equal in the two runs. The full security proof can be found in Appendix B.2.

5.5 Case Study

Now we resume the analysis of HotCRP, started in section 5.2 where we described the login CSRF and proposed a fix, and describe the remaining session integrity problems we discovered by typing its model in our core calculus. The encodings of Moodle and phpMyAdmin, including the description of the new vulnerability, are provided in Appendix B.1.6.

5.5.1 Methodology

We type-check the HotCRP model of section 5.2 against different attackers, including the web-, related-domain-, and network attacker. Two scenarios motivate the importance of the related-domain attacker in our case study. First, many conferences using HotCRP deploy the system on a subdomain of the university organizing the event, e.g., CSF 2020: any user who can host contents on a subdomain of the university can act as the attacker. Second, anybody can host a conference on a subdomain of `hotcrp.com` or access the administrative panel of `test.hotcrp.com`: by exploiting a stored XSS vulnerability (now fixed) in the admin panel, it was possible to show on the homepage of the conference a message containing JavaScript code that tampers with cookies to implement the attacks below.

Failures in type-checking highlight code portions that we analyze manually, as they likely suffer from session integrity flaws. Once a problem is identified, we implement a patch in our HotCRP model and try to type-check it again; this iterative process stops when we manage to establish a security proof by typing, as shown in subsection 5.5.3.

5.5.2 Cookie Integrity Attacks

Our fix against login CSRF does not ensure the integrity of session cookies against network and related-domain attackers: the former can compromise cookie integrity by forging HTTP traffic, while the latter can set cookies for the target website by using the `Domain` attribute. Attackers can thus perform *cookie forcing* to set the their session cookies in the victim's browser, achieving the same outcome of a login CSRF.

Even worse, the lack of cookie integrity combined with a logical vulnerability on HotCRP code enables a *session fixation* attack, where the attacker manages to force a known cookie into the browser of the victim before she authenticates which is used by HotCRP to identify the victim's session after login. With the known cookie, the attacker can then access the victim's session to steal submitted papers, send fake reviews, or deanonymize reviewers. HotCRP tries to prevent session fixation by checking during login whether the provided session cookie (if any) identifies a session where no variable is set: in such a case, the value of the cookie is changed to an unpredictable random string. However, some session variables are not properly unset during logout, thus the above check can be voided by an attacker with an account on the target website that obtains a valid cookie by authenticating and logging out.[5] At this point, the attacker can inject this cookie into the victim's browser to perform the attack.

Both attacks are captured in typing as follows: although we have a certain liberty in the choice of our initial environment, no possible type for sid leads to a successful type derivation since sid must have a credential type. As the attacker can set the cookie, it must have low integrity by well-formedness of the typing environment (Definition 3). Since the attacker can write (low confidentiality) values of her knowledge into sid, it may not be a credential of high confidentiality, again by Definition 3. Hence we must assume that sid is a credential of low confidentiality and integrity. However, since the user's password has high confidentiality, typing fails in the *login* endpoint (on line 9) when applying rule (T-LOGIN).

A possible solution against these threats relies on the adoption of *cookie prefixes* (cf. subsection 5.3.3) which provide high integrity guarantees against network and related-domain attackers. This protection cannot be applied by default in HotCRP due to backward compatibility reasons, i.e., `hotcrp.com` relies on cookies shared across multiple domains to link different conferences under the same account. However, the developer has fixed the bug causing the session fixation vulnerability and we have discussed with him the option to offer cookie prefixes as an opt-in security mechanism during the setup of HotCRP.

[5] To simplify the presentation, this complex behavior is not encoded in the example in section 5.2. However, the possibility to perform cookie forcing, which is modeled in our example, is a prerequisite for session fixation and is detected by the type system.

5.5.3 Typing Example

Now we show how to type-check the fixed *login* endpoint (from subsection 5.2.4) on domain d_C against an attacker controlling a related-domain $d_E \sim d_C$, assuming that the session cookie is secured with the `__Host-` prefix. We let the attacker label $\ell_a = (\mathsf{http}(d_E) \vee \mathsf{https}(d_E), \mathsf{http}(d_E) \vee \mathsf{https}(d_E))$, and let $\ell_C = (\mathsf{https}(d_C), \mathsf{https}(d_C))$, $\ell_{LH} = (\bot_C, \mathsf{https}(d_C))$, $\ell_{HL} = (\mathsf{https}(d_C), \top_I)$. We then consider a minimal environment Γ sufficient to type the *login* endpoint, where:

$$\begin{aligned}
\Gamma_{\mathcal{U}} &= \{\mathit{login} \mapsto (\ell_C, (\ell_{LH}, \mathtt{cred}(\ell_C), \mathtt{cred}(\ell_{HL})), \mathsf{https}(d_C)),\\
&\qquad \mathit{manage} \mapsto (\ell_C, (\ell_C, \ell_{LH}, \mathtt{cred}(\ell_{HL})), \mathsf{https}(d_C))\}\\
\Gamma_{\mathcal{R}^{@}} &= \{r \mapsto \mathtt{cred}(\ell_C), r' \mapsto \mathtt{cred}(\ell_{HL}),\\
&\qquad \mathit{sid} \mapsto \mathtt{cred}(\ell_C), \mathit{pre} \mapsto \mathtt{cred}(\ell_{HL})\}\\
\Gamma_{\mathcal{R}^{\$}} &= \{\mathit{user} \mapsto \ell_{LH}, \mathit{ltoken} \mapsto \mathtt{cred}(\ell_{HL})\}\\
\Gamma_{\mathcal{V}} &= \{\mathtt{auth} \mapsto \Gamma_{\mathcal{U}}(\mathit{login}), \mathtt{link} \mapsto \Gamma_{\mathcal{U}}(\mathit{manage})\}
\end{aligned}$$

We type-check the code under two different assumptions in (T-RECV). Our goal is to prune the CSRF typing branch before the security critical part and type it only in the honest setting.

We start with the honest typing branch. When typing the conditional (line 2) in rule (T-IF), we do not lower $\mathtt{pc}$ since the integrity label of the guard and $\mathtt{pc}$ is $\mathsf{https}(d_C)$. In the **then** branch (line 3), we have the assignment $@r' := \mathit{fresh}()^{\ell_{HL}}$, which types successfully according to (T-SETGLOBAL).[6] The **start** statement with the freshly sampled value yields a session label $\ell_s = (\mathsf{https}(d_C), \top_I)$. The assignment $\$\mathit{ltoken} := \mathit{fresh}()^{\ell_{HL}}$ also succeeds according to (T-SETSESSION). The session label does not affect the type of the reference $\$\mathit{ltoken}$ in this case. For the **reply** (lines 4–6) we successfully check that the URL is well-formed and may be produced with the current $\mathtt{pc}$ (T-FORM), that the empty script is well-typed, and that $y = @r'$ may be assigned to the cookie pre (T-REPLY). In the **else** branch of the conditional, we start a session over the cookie $@\mathit{pre}$ (line 8), leading to a session label $\ell_s = (\mathsf{https}(d_C), \top_I)$ (T-START). The conditions in (T-TCHK) are fulfilled for the **tokenchk** command (line 9) and we continue typing without any additional effect. Since we still have $\mathtt{pc} = \mathsf{https}(d_C)$, the assignment $@r := \mathit{fresh}()^{\ell_C}$ type-checks (line 10). As the password is of the same type as the reference $@r$ containing the session secret, the **login** also type-checks successfully (T-LOGIN). The **start** statement over a credential of type $\mathtt{cred}(\ell_C)$ gives us the session label $\ell_s = \ell_C$ (line 11). For the **reply** (lines 12–14), we check that we may include the form with the current $\mathtt{pc}$ and that it is well formed (trivial since it contains only $\bot$), that the empty script is well-typed and that we may assign the value of $@r$ to the cookie sid (T-REPLY).

The **then** branch of the CSRF case types similarly to the honest case, since all references used in it and the cookie pre have integrity label $\top_I$. Additionally, in the CSRF branch, we do not type the DOM (T-REPLY). In the **else** branch we start a session (line 8) with label $\ell_s = (\mathsf{https}(d_C), \top_I)$ (T-START). When performing the **tokenchk** (line 9), we can apply rule (T-PRUNETCHK), since $\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \$\mathit{ltoken} : \mathtt{cred}(\ell_{HL})$ and $\Gamma, \ell_s \vdash^{\mathsf{se}}_{\ell_a} \mathit{token} : \ell_a$ cannot be given the same confidentiality label. Hence, we do not have to type-check the continuation.

[6] Here we expose the annotations of *fresh()* expressions (needed for typing) that we omitted from section 5.2 for readability purposes.

5.6 Related Work

Formal foundations for web security have been proposed in a seminal paper [ABL+10], using a model of the web infrastructure expressed in the Alloy model-checker to find violations of expected web security goals. Since then, many other papers explored formal methods in web security: a recent survey [BCF17] covers different research lines. We discuss here the papers which are closest to our work.

In the context of web sessions, [BCFK15] employed *reactive non-interference* [BPS+09] to formalize and prove strong confidentiality properties for session cookies protected with the `HttpOnly` and `Secure` attributes, a necessary condition for any reasonable notion of session integrity. A variant of reactive non-interference was also proposed in [KCB+14] to formalize an integrity property of web sessions which rules out CSRF attacks and malicious script inclusions. The paper also introduced a browser-side enforcement mechanism based on *secure multi-execution* [DP10]. A more general definition of web session integrity, which we adapted in the present paper, was introduced in [BCF+14] to capture additional attacks, like password theft and session fixation. The paper also studied a provably sound browser-based enforcement mechanism based on runtime monitoring. Finally, [CFGM16a] proposed the adoption of *micro-policies* [dADG+15] in web browsers to prevent a number of attacks against web sessions and presented Michrome, a Google Chrome extension implementing the approach. None of these papers, however, considered the problem of enforcing a formal notion of session integrity by analyzing web application code, since they only focused on browser-side defenses.

Formal methods found successful applications to web session security through the analysis of *web protocols*, which are the building blocks of web sessions when single sign-on services are available. Bounded model-checking was employed in [ACC+08] and [ACC+13] to analyze the security of existing single sign-on protocols, exposing real-world attacks against web authentication. WebSpi is a ProVerif library designed to model browser-server interactions, which was used to analyze existing implementations of single sign-on based on OAuth 2.0 [BBDM14] and web-based cloud providers [BBDM13].

Web protocols for single sign-on have also been manually analyzed in the expressive Web Infrastructure Model (WIM): for instance, [FKS16] focused on OAuth 2.0, [FKS17] considered OpenID Connect, and [FHK19] analyzed the OpenID Financial-grade API. While the WIM is certainly more expressive than our core model, proofs are at present manual and require a strong human expertise. In terms of security properties, [FHK19] considers a session integrity property expressed as a trace property that is specific to the OpenID protocol flow and the resources accessed thereby, while our definition of session integrity is generic and formulated as a hyperproperty.

Server-side programming languages with formal security guarantees have been proposed in several research papers. Examples include SELinks [CSH09], UrFlow [Chl10], SeLINQ [SHS14] and JSLINQ [BLSS16]. All these languages have the ability to enforce information flow control in multi-tier web applications, potentially including a browser, a server and a database. Information flow control is an effective mechanism to enforce session integrity, yet these papers do not discuss how to achieve web session security; rather, they propose new languages and abstractions for

developing web applications. To the best of our knowledge, there is no published work on the formal security analysis of server-side programming languages, though the development of accurate semantics for such languages [FM14] is undoubtedly a valuable starting point for this kind of research.

5.7 Conclusion

We introduced a type system for sound verification of session integrity for web applications encoded in a core model of the web, and used it to assess the security of the session management logic of HotCRP, Moodle, and phpMyAdmin. During this process we unveiled novel critical vulnerabilities that we responsibly disclosed to the applications' developers, validating by typing the security of the fixed versions.

We are currently developing a type-checker to fully automate the analysis, which we intend to make available as open source. Providing type annotations is typically straightforward, as they depend on the web application specification and are easily derivable from it (e.g., cookie labels are derived from their attributes) and typing derivations are mostly deterministic, with a few exceptions (e.g., subtyping) that however follow recurrent patterns (e.g., subtyping is used in assignments to upgrade the value type to the reference type).

Furthermore, while in this work we focused on a concise web model to better illustrate the foundational aspects of our analysis technique, it would be interesting to extend the type system to cover richer web models, e.g., the WIM model [FHK19], as well as additional web security properties. We also plan to automate the verification process for PHP code, e.g., by developing an automated translation from real world code into our calculus. Finally, we would like to formalize our theory in a proof assistant.

Acknowledgments

This work has been partially supported by the the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC); by the Austrian Science Fund (FWF) through the project PROFET (grant agreement P31621); by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 SBA.

CHAPTER 6

A Monadic Framework for Relational Verification

Applied to Information Security, Program Equivalence, and Optimizations

Abstract

Relational properties describe multiple runs of one or more programs. They characterize many useful notions of security, program refinement, and equivalence for programs with diverse computational effects, and they have received much attention in the recent literature. Rather than developing separate tools for special classes of effects and relational properties, we advocate using a general purpose proof assistant as a unifying framework for the relational verification of effectful programs. The essence of our approach is to model effectful computations using monads and to prove relational properties on their monadic representations, making the most of existing support for reasoning about pure programs.

We apply this method in F* and evaluate it by encoding a variety of relational program analyses, including information flow control, program equivalence and refinement at higher order, correctness of program optimizations and game-based cryptographic security. By relying on SMT-based automation, unary weakest preconditions, user-defined effects, and monadic reification, we show that, compared to unary properties, verifying relational properties requires little additional effort from the F* programmer.

This chapter presents the result of a research project that was started during my internship with Cédric Fournet at Microsoft Research, Cambridge. The result, a collaboration with Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy and Santiago Zanella-Béguelin, has been published at the 7th ACM

SIGPLAN International Conference on Certified Programming and Proofs (CPP'18) [GMF+18]. I contributed through preliminary experiments with different approaches of performing relational reasoning in F⋆ and am responsible for the results reported on information flow control in Section 6.5.

6.1 Introduction

Generalizing unary properties (which describe single runs of programs), *relational* properties describe multiple runs of one or more programs. Relational properties are useful when reasoning about program refinement, approximation, equivalence, provenance, as well as many notions of security. A great many relational program analyses have been proposed in the recent literature, including [Yan07, ZP08, BKBH09, KTL09, GS10, BGZ12, BKOZ13, BFG+14, BGA+15, HS12, KTB+15, BNN16, ASK16, cCLRR16, AGH+17, BKU15, BBG+17, MMB+13, FC16, BPPR16, BPPR17] and [ÇBG+17]. While some systems have been designed for the efficient verification of specialized relational properties of programs (notably information-flow type systems, e.g., [SM03]), others support larger classes of properties. These include tools based on product program constructions for automatically proving relations between first-order imperative programs (e.g., SymDiff [LHKR12] and Descartes [SD16]), as well as relational program logics [Ben04] that support interactive verification of relational properties within proof assistants (e.g., EasyCrypt [BGZ12] and RHTT [NBG13]).

We provide a framework in which relational logics and other special-purpose tools can be recast on top of a general method for relational reasoning. The method is simple: we use monads to model and program effectful computations; and we reveal the pure monadic representation of an effect in support of specification and proof. Hence, we reduce the problem of relating effectful computations to relating their pure representations, and then apply the tools available for reasoning about pure programs. While this method should be usable for a variety of proof assistants, we choose to work in F⋆ [SHK+16], a dependently typed programming language and proof assistant. By relying on its support for SMT-based automation, unary weakest preconditions, and user-defined effects [AHM+17], we demonstrate, through a diverse set of examples, that our approach enables the effective verification of relational properties with an effort comparable to proofs of unary properties in F⋆ and to proofs in relational logics with SMT-based automation.

Being based on an expressive semantic foundation, our approach can be directly used to verify relational properties of programs. Additionally, we can still benefit from more specialized automated proof procedures, such as syntax-directed relational type systems, by encoding them within our framework. Hence, our approach facilitates comparing and composing special-purpose relational analyses with more general-purpose semi-interactive proofs; and it encourages prototyping and experimenting with special-purpose analyses with a path towards their certified implementations.

6.1.1 A first example

We sketch the main ideas on a proof of equivalence for the two stateful, recursive functions below, a task not easily accomplished using specialized relational program logics:

```
let rec sum_up r lo hi =
  if lo≠hi then (r := !r+lo; sum_up r (lo+1) hi)
let rec sum_dn r lo hi =
  if lo≠hi then (r := !r+hi−1; sum_dn r lo (hi−1))
```

Both functions sum all numbers between lo and hi into some accumulator reference r, the former function by counting up and the latter function by counting down.

Unary reasoning about monadic computations As a first step, we embed these computations within a dependently typed language. There are many proposals for how to do this—one straightforward approach is to encapsulate effectful computations within a parameterized monad [Atk09]. In F⋆, as in the original Hoare Type Theory [NMB08], these monads are indexed by a computation's pre- and postconditions and proofs are conducted using a unary program logic (i.e., not relational), adapted for use with higher-order, dependently typed programs. Beyond state, F⋆ supports reasoning about unary properties of a wide class of user-defined monadic effects, where the monad can be chosen to best suit the intended style of unary proof.

Relating reified effectful terms Our goal is to conveniently state and prove properties that relate effectful terms, e.g., prove sum_up and sum_dn equivalent. We do so by revealing the monadic representation of these two computations as pure state-passing functions. However, since doing this naïvely would preclude the efficient implementation of primitive effects, such as state in terms of a primitive heap, our general method relies on an explicit *monadic reification* coercion for exposing the pure monadic representation of an effectful computation in support of relational reasoning.[1] Thus, in order to relate effectful terms, one simply reasons about their pure reifications. Turning to our example, we prove the following lemma, stating that running sum_up and sum_dn in the same initial states produces equivalent final states. (A proof is given in §6.2.4.)

```
r:ref int → lo:int → hi:int{hi ≥ lo} → h:heap{r ∈ h} →
  reify (sum_up r lo hi) h ∼ reify (sum_dn r lo hi) h
```

Flexible specification and proving style with SMT-backed automation Although seemingly simple, proving sum_up and sum_dn equivalent is cumbersome, if at all possible, in most prior relational program logics. Prior relational logics rely on common syntactic structure and control flow between multiple programs to facilitate the analysis. To reason about transformations like loop reversal, rules exploiting syntactic similarity are not very useful and instead a typical proof in prior systems may involve several indirections, e.g., first proving the full functional correctness of each loop with respect to a purely functional specification and then showing that the two

[1] While this coercion is inspired by [Fil94] reify operator, we only use it to reveal the pure representation of an effectful computation in support of specification and proof, whereas Filinski's main use of reification was to uniformly implement monads using continuations.

specifications are equivalent. Through monadic reification, effectful terms are *self-specifying*, removing the need to rewrite the same code in purely-functional style just to enable specification and reasoning.

Further, whereas many prior systems are specialized to proving binary relations, it can be convenient to structure proofs using relations of a higher arity, a style naturally supported by our method. For example, a key lemma in the proof of the equivalence above is an inductive proof of a ternary relation, which states that sum_up is related to sum_up on a prefix combined with sum_dn on a suffix of the interval [lo, hi).

Last but not least, using the combination of typechecking, weakest precondition calculation, and SMT solving provided by F*, many relational proofs go through with a degree of automation comparable to existing proofs of unary properties, as highlighted by the examples in this paper.

6.1.2 Contributions and outline

We propose a methodology for relational verification (§6.2), covering both broadly applicable ingredients such as representing effects using monads and exposing their representation using monadic reification, as well as our use of specific F* features that enable proof flexibility and automation. All these ingredients are generic, i.e., none of them is specific to the verification of relational properties.

The rest of the paper is structured as a series of case studies illustrating our methodology at work. Through these examples we aim to show that our methodology enables comparing and composing various styles of relational program verification in the same system, thus taking a step towards unifying many prior strands of research. Also these examples cover a wide range of applications that, when taken together, exceed the ability of all previous tools for relational verification of which we are aware. Our examples are divided into three sections that can be read in any order, each being an independent case study:

Transformations of effectful programs (§6.3) We develop an extensional, semantic characterization of a stateful program's read and write effects, based on the relational approach of [BKHB06]. Based on these semantic read and write effects, we derive lemmas that we use to prove the correctness of common program transformations, such as swapping the order of two commands and eliminating redundant writes. Going further, we encode [Ben04] relational Hoare logic in our system, providing a syntax-directed proof system for relational properties as a special-purpose complement to directly reasoning about a program's effects.

Cryptographic security proofs (§6.4) We show how to model basic game steps of code-based cryptographic proofs of security [BR06] by proving equivalences between probabilistic programs. We prove perfect secrecy of one-time pad encryption , an elementary use of [BGZ09] probabilistic relational Hoare logic.

Information-flow control (§6.5) We encode several styles of static information-flow control analyses . Highlighting the ability to compose various proof styles in a single framework, we combine automated, type-based security analysis with SMT-backed, semantic proofs of noninterference.

Proofs of algorithmic optimizations (§6.6) With a few exceptions, prior relational program logics apply to first-order programs and provide incomplete proof rules that exploit syntactic similarities between the related programs. Not being bound by syntax, we prove relations of higher arities (e.g., 4-ary and 6-ary relations) between higher-order, effectful programs with differing control flow by reasoning directly about their reifications. We present two larger examples: First, we show how to memoize a recursive function using [McB15] partiality monad and we prove it equivalent to the original non-memoized version. Second, we implement an imperative union-find data structure, adding the classic union-by-rank and path compression optimizations in several steps and proving stepwise refinement.

From these case studies, we conclude that our method for relational reasoning about reified monadic computations is both effective and versatile. We are encouraged to continue research in this direction, aiming to place proofs of relational properties of effectful programs on an equal footing with proofs of pure programs in F* as well as other proof assistants and verification tools.

The code for the examples in this paper is available at
https://github.com/FStarLang/FStar/tree/master/examples/rel
Compared to this code, the listings in the paper are edited for clarity and sometimes omit uninteresting details. The extended version [GMF$^+$18] describes some additional case studies that we omit here because of space.

6.2 Methodology for relational verification

In this section we review in more detail the key F* features we use and how each of them contributes to our verification method for relational properties. Two of these features are general and broadly applicable: (§6.2.1) modeling effects using monads and keeping the effect representation abstract to support efficient implementation of primitive effects and (§6.2.3) using monadic reification to expose the effect representation. The remaining features are more specific to F* and enable proof flexibility and automation: (§6.2.2) using a unary weakest precondition calculus to produce verification conditions in an expressive dependently typed logic; (§6.2.4) using dependent types together with pre- and postconditions to express arbitrary relational properties of reified computations; (§6.2.4) embedding the dependently typed logic into SMT logic to enable the SMT solver to reason by computation.

None of these generic ingredients is tailored to the verification of relational properties, and while F* is currently the only verification system to provide all these ingredients in a unified package, each of them also appears in other systems. This makes us hopeful that this relational verification method can also be applied with other proof assistants (e.g., Coq, Lean, Agda, Idris, etc.), for which the automation would likely come in quite different styles.

6.2.1 Modeling effects using monads

At the core of F* is a language of dependently typed, total functions. Function types are written x:t $\rightarrow$ Tot t' where the co-domain t' may depend on the argument x:t. Since it is the default in F*, we often drop the Tot annotation (except where needed for emphasis) and also the name of the

formal argument when it is unnecessary, e.g., we write int → bool for _:int → Tot bool. We also write #x:t → t' to indicate that the argument x is implicitly instantiated.

Our first step is to describe effects using monads built from total functions [Mog89]. For instance, here is the standard monadic representation of state in F⋆ syntax.

```
type st (mem:Type) (a:Type) = mem → Tot (a ∗ mem)
```

This defines a type st parameterized by types for the memory (mem) and the result (a). We use st as the representation type of a new STATE_m effect we add to F⋆, with the total qualifier enabling the termination checker for STATE_m computations.

```
total new_effect {
 STATE_m (mem:Type) : a:Type → Effect
 with repr = st mem;
  return = λ(a:Type) (x:a) (m:mem) → x, m;
  bind = λ(a b:Type) (f:st mem a) (g:a → st mem b) (m:mem) →
   let z, m' = f m in g z m';
  get = λ() (m:mem) → m, m; put = λ(m:mem) _ → (), m }
```

This defines the return and bind of this monad, and two actions: get for obtaining the current memory, and put for updating it. The new effect STATE_m is still parameterized by the type of memories, which allows us to choose a memory model best suited to the programming and verification task at hand. We often instantiate mem to heap (a map from references to their values, as in ML), obtaining the STATE effect shown below—we use other memory types in §6.5 and §6.6.

```
total new_effect STATE = STATE_m heap
```

While such monad definitions could in principle be used to directly extend the implementation of any functional language with the state effect, a practical language needs to allow keeping the representation of some effects abstract so that they are efficiently implemented primitively [Pey10]. F⋆ uses its simple module system to keep the monadic representation of the STATE effect abstract and implements it under the hood using the ML heap, rather than state passing (and similarly for other primitive ML effects such as exceptions). Whether implemented primitively or not, the monadic definition of each effect is always the *model* used by F⋆ to reason about effectful code, both intrinsically using a (non-relational) weakest precondition calculus (§6.2.2) and extrinsically using monadic reification (§6.2.3).

For the purpose of verification, monads provide great flexibility in the modeling of effects, which enables us to express relational properties and to conduct proofs at the right level of abstraction. For instance, in §6.4 we define a monad for random sampling from a uniform distribution, and in §6.6.1 we define a partiality monad for memoizing recursive functions. Moreover, since the difficulty of reasoning about effectful code is proportional to the complexity of the effect, we do not use a single full-featured monad for all code; instead we define custom monads for sub-effects and relate them using monadic lifts. For instance, we define a READER monad for computations that only read the store, lifting READER to STATE only where necessary (§6.5.1 provides a

detailed example). While F⋆ code is always written in an ML-like direct style, the F⋆ typechecker automatically inserts binds, returns and lifts under the hood [SGLH11].

6.2.2 Unary weakest preconditions for user-defined effects and intrinsic proof

For each user-defined effect, F⋆ derives a weakest precondition calculus for specifying unary properties and computing verification conditions for programs using that effect [AHM+17]. Each effect definition induces a computation type indexed by a predicate transformer describing that computation's effectful semantics.

For state, we obtain a computation type 'STATE a wp' indexed by a result type a and by wp, a predicate transformer of type (a → heap → Type) → heap → Type, mapping postconditions (relating the result and final state of the computation) to preconditions (predicates on the initial state). The types of the get and put actions of STATE are specified as:

```
val get : unit → STATE heap (λ post (h:heap) → post h h)
val put : h':heap → STATE unit (λ post (h:heap) → post () h')
```

The type of get states that, in order to prove any postcondition post of 'get ()' evaluated in state h, it suffices to prove post h h, whereas for put h' it suffices to prove post () h'. F⋆ users find it more convenient to index computations with pre- and postconditions as in HTT [NMB08], or sometimes not at all, using the following abbreviations:

```
ST a (requires p) (ensures q) = STATE a (λ post h0 →
                p h0 ∧ (∀ (x:a) (h1:heap). q h0 x h1 ⟹ post x h1))
St a = ST a (requires (λ _ → ⊤)) (ensures (λ _ _ _ → ⊤))
```

F⋆ computes weakest preconditions generically for any effect. Intuitively, this works by putting the code into an explicit monadic form and then translating the binds, returns, actions, and lifts from the expression level to the weakest precondition level. This enables a convenient form of *intrinsic* proof in F⋆, i.e., one annotates a term with a type capturing properties of interest; F⋆ computes a weakest precondition for the term and compares it to the annotated type using a built-in subsumption rule, checked by an SMT solver.

For example, the sum_up function from §6.1.1 can be given the following type:

```
r:ref int → lo:nat → hi:nat{hi ≥ lo} →
            ST unit (requires λh → r ∈ h) (ensures λ_ _ h → r ∈ h)
```

This is a dependent function type, for a function with three arguments r, lo, and hi returning a terminating, stateful computation. The *refinement* type hi:nat{hi ≥ lo} restricts hi to only those natural numbers greater than or equal to lo. The computation type of 'sum_up r lo hi' simply requires and ensures that its reference argument r is present in the memory. F⋆ computes a weakest precondition from the implementation of sum_up (using the types of (!) and (:=) provided by the heap memory model used by STATE) and proves that its inferred specification is subsumed by the user-provided annotation. The same type can also be given to sum_dn.

6.2.3 Exposing effect definitions via reification

Intrinsic proofs of effectful programs in F⋆ are inherently restricted to unary properties. Notably, pre- and postconditions are required to be pure terms, making it impossible for specifications to refer directly to effectful code, e.g., sum_up cannot directly use itself or sum_dn in its specification. To overcome this restriction, we need a way to coerce a terminating effectful computation to its underlying monadic representation which is a pure term—[Fil94] monadic reification provides just that facility.[2]

Each new effect in F⋆ induces a reify operator that exposes the representation of an effectful computation in terms of its underlying monadic representation [AHM+17]. For the STATE effect, F⋆ provides the following (derived) rule for reify, to coerce a stateful computation to a total, explicitly state-passing function of type heap → t ∗ heap. The argument and result types of reify e are refined to capture the pre- and postconditions intrinsically proved for e.

$$\frac{S;\Gamma| - e : \mathsf{ST}\ \mathsf{t}\ (\mathsf{requires}\ \mathsf{pre})\ (\mathsf{ensures}\ \mathsf{post})}{S;\Gamma| - \mathsf{reify}\ e : \mathsf{h{:}heap}\{\mathsf{pre\ h}\} \to \mathsf{Tot}\ (\mathsf{r{:}(t{*}heap)}\{\mathsf{post\ h\ (fst\ r)\ (snd\ r)}\})}$$

The semantics of reify is to traverse the term and to gradually expose the underlying monadic representation. We illustrate this below for STATE, where the constructs on the right-hand side of the rules are the pure implementations of return, bind, put, and get as defined on page 136, but with type arguments left implicit:

$$\begin{aligned}
\mathsf{reify}\ (\mathsf{return}\ e) &\rightsquigarrow \mathsf{STATE.return}\ e \\
\mathsf{reify}\ (\mathsf{bind\ x} \leftarrow e_1\ \mathsf{in}\ e_2) &\rightsquigarrow \mathsf{STATE.bind}\ (\mathsf{reify}\ e_1)(\lambda \mathsf{x} \rightarrow \mathsf{reify}\ e_2) \\
\mathsf{reify}\ (\mathsf{get}\ e) &\rightsquigarrow \mathsf{STATE.get}\ e \\
\mathsf{reify}\ (\mathsf{put}\ e) &\rightsquigarrow \mathsf{STATE.put}\ e
\end{aligned}$$

Armed with reify, we can write an *extrinsic* proof of a lemma relating sum_up and sum_dn (discussed in detail in §6.2.4), i.e., an "after the fact" proof that is separate from the definition of sum_up and sum_dn and that relates their reified executions. We further remark that in F⋆ the standard operational semantics of effectful computations is modeled in terms of reification, so proving a property about a reified computation is really the same as proving the property about the evaluation of the computation itself.

The reify operator clearly breaks the abstraction of the underlying monad and needs to be used with care. [AHM+17] show that programs that do not use reify (or its converse, reflect) can be compiled efficiently. Specifically, if the computationally relevant part of a program is free of reify then the STATE computations can be compiled using primitive state with destructive updates.

To retain these benefits of abstraction, we rely on F⋆'s module system to control how the abstraction-breaking reify coercion can be used in client code. In particular, when abstraction violations cannot be tolerated, we use F⋆'s Ghost effect (explained in §6.2.4) to mark reify as being usable only in computationally irrelevant code, limiting the use of monadic reification to specifications and proofs. This allows one to use reification even though effects like state and exceptions are implemented primitively in F⋆.

[2] Less frequently, we use `reify`'s dual, `reflect`, which packages a pure function as an effectful computation.

6.2.4 Extrinsic specification and proof, eased by SMT-based automation

We now look at the proof relating sum_up and sum_dn in detail, explaining along the way several F*-specific idioms that we find essential to making our method work well.

Computational irrelevance (Ghost effect) The Ghost effect is used to track a form of computational irrelevance. Ghost t (requires pre) (ensures post) is the type of a pure computation returning a value of type t satisfying post, provided pre is valid. However, this computation must be erased before running the program, so it can only be used in specifications and proofs.

Adding proof irrelevance (Lemma) F* provides two closely related forms of proof irrelevance. First, a pure term e:t can be given the refinement type x:t{ϕ} when it validates the formula ϕ[e/x], although no proof of ϕ is materialized. For example, borrowing the terminology of [Nog02], the value () is a *squashed* proof of u:unit{$0 \leq 1$}. Combining proof and computation irrelevance, e : Ghost unit pre ($\lambda() \rightarrow$ post) is a squashed proof of pre $\rightarrow$ post. This latter form is so common that we write it as Lemma (requires pre) (ensures post), further abbreviated as Lemma post when pre is $\top$.

Proof relating sum_up and sum_dn Spelling out the main lemma of §6.1.1, our goal is a value of the following type:

```
val eq_sum_up_dn (r:ref int)(lo:int)(hi:int{hi ≥ lo})(h:heap{r ∈ h})
: Lemma
  (v r (reify (sum_up r lo hi) h) == v r (reify (sum_dn r lo hi) h))
```

where v r (_, h) = h.[r] and h.[r] selects the contents of the reference r from the heap h.

An attempt to give a trivial definition for eqsum_up_dn that simply returns a unit value () fails, because the SMT solver cannot automatically prove the strong postcondition above. Instead our proof involves calling an auxiliary lemma sum_up_dn_aux, proving a ternary relation:

```
val sum_up_dn_aux (r:ref int) (lo:int) (mid:int{mid ≥ lo})
                  (hi:int{hi ≥ mid}) (h:heap{r ∈ h})
: Lemma (v r (reify (sum_up r lo hi) h)
          == v r (reify (sum_dn r lo mid) h)
            + v r (reify (sum_up r mid hi) h) − h.[r])
  (decreases (mid − lo))
let eq_sum_up_dn r lo hi h = sum_up_dn_aux r lo hi hi h
```

While the statement of eq_sum_up_dn is different from the statement of sum_up_dn_aux, the SMT-based automation fills in the gaps and accepts the proof sketch. In particular, the SMT solver figures out that sum_up r hi hi is a no-op by looking at its reified definition. In other cases, the user has to provide more interesting proof sketches that include not only calls to lemmas that the SMT solver cannot automatically apply but also the cases of the proof and the recursive structure. This is illustrated by the following proof:

```
let rec sum_up_dn_aux r lo mid hi h =
```

```
if lo ≠ mid then (sum_up_dn_aux r lo (mid − 1) hi h;
                  sum_up_commute r mid hi (mid − 1) h;
                  sum_dn_commute r lo (mid − 1) (mid − 1) h)
```

This proof is by induction on the difference between mid and lo (as illustrated by the decreases clause of the lemma, this is needed because we are working with potentially-negative integers). If this difference is zero, then the property is trivial since the SMT solver can figure out that sum_dn r lo lo is a no-op. Otherwise, we call sum_up_dn_aux recursively for mid − 1 as well as two further commutation lemmas (not shown) about sum_up and sum_dn and the SMT automation can take care of the rest.

Encoding computations to SMT So how did F⋆ figure out automatically that sum_up r hi hi and sum_dn r lo lo are no-ops? For a start the F⋆ normalizer applied the semantics of reify sketched in §6.2.3 to partially evaluate the term and reveal the monadic representation of the STATE effect by traversing the term and unfolding the monadic definitions of return, bind, actions and lifts. In the case of reify (sum_up r hi hi) h, for instance, reduction intuitively proceeds as follows:

```
reify (sum_up r hi hi) h
  ⇝ reify (if hi ≠ hi then (r := !r + lo; sum_up r (lo + 1) hi)) h
  ⇝* if hi ≠ hi then (STATE.bind (reify (Ref.read r) h) (λ x →
                        STATE.bind (reify (Ref.upd r (x + lo)))
                          (λ _ → reified_sum_up r (hi + 1) hi))) h
       else STATE.return () h
  ⇝* if hi ≠ hi then let x, h' = reify (Ref.read r) h in
                        let _, h'' = reify (Ref.upd r (x + lo)) h' in
                        reified_sum_up r (hi + 1) hi h''
       else ((), h)
```

What is left is pure monadic code that F⋆ then encodes to the SMT solver in a way that allows it to reason by computation [AHKS16]. For reify (sum_up r hi hi) h the SMT solver can trivially show that hi ≠ hi is false and thus the computation returns the pair ((), h).

While our work did not require any extension to F⋆'s theory [AHM+17], we significantly improved F⋆'s logical encoding to perform normalization of open terms based on the semantics of reify (a kind of symbolic execution) before calling the SMT solver. This allowed us to scale and validate the theory of [AHM+17] from a single 2-line example to the ≈4,300 lines of relationally verified code presented in this paper.

6.2.5 Empirical evaluation of our methodology

For this first example, we reasoned directly about the semantics of two effectful terms to prove their equivalence. However, we often prefer more structured reasoning principles to prove or enforce relational properties, e.g., by using program logics, syntax-directed type systems, or even dynamic analyses. In the rest of this paper, we show through several case studies, that these approaches can be accommodated, and even composed, within our framework.

Subject	Section	1st run (ms)	Replay (ms)	Loc
Loops	6.1.1	218192	8943	127
Reorderings	6.3.1	9239	4749	158
Benton (2004)	6.3.2	832706	22920	1352
Cryptography	6.4	17307	10015	530
Static IFC	6.5.1	68525	15909	730
Hybrid IFC	6.5.2	55472	1038	34
Declassification	*	63763	9811	208
IFC Monitor	*	44589	11480	502
Memoization	6.6.1	12198	12294	427
Union-find	6.6.2	89838	33455	295
Total		1411829	130614	4363

Table 6.1: Code size (lines of code without comments) and proof-checking time (ms) for our examples. Examples with label * appear in the extended version [GMF+18].

Table 6.1 summarizes the empirical evaluation from these case studies. Each row describes a specific case study, its size in lines of source code, and the verification time using F⋆ and the Z3-4.5.1 SMT solver. The verification times were collected on an Intel Xeon E5-2620 at 2.10 GHz and 32GB of RAM. The "1st run" column indicates the time it takes F⋆ and Z3 to find a proof. This proof is then used to generate hints (unsat cores) that can be used as a starting point to verify subsequent versions of the program. The "replay" column indicates the time it takes to verify the program given the hints recorded in the first run. Proof replay is usually significantly faster, indicating that although finding a proof may initially be quite expensive, revising a proof with hints is fast, which greatly aids interactive proof development.

6.3 Correctness of program transformations

Several researchers have devised custom program logics for verifying transformations of imperative programs [Ben04, BGZ09, CKMR12]. We show how to derive similar rules justifying the correctness of generic program transformations within our monadic framework. We focus on stateful programs with a fixed-domain, finite memory. We leave proving transformations of commands that dynamically allocate memory to future work.

6.3.1 Generic transformations based on read- and write-footprints

Here and in the next subsection, we represent a command c as a function of type unit → St unit that may read or write arbitrary references in memory.

```
type command = unit → St unit
```

In trying to validate transformations of commands, it is traditional to employ an effect system to delimit the parts of memory that a command may read or write. Most effect systems are unary,

syntactic analyses. For example, consider the classic frame rule from separation logic:

$$\{P\}c\{Q\} \Rightarrow \{P * R\}c\{Q * R\}$$

The command c requires ownership of a subset of the heap P in order to execute, then returns ownership of Q to its caller. Any distinct heap fragment R remains unaffected by the function. Reading this rule as an effect analysis, one may conclude that c may read or write the P-fragment of memory—however, this is just an approximation of c's extensional behavior. [BKHB06] observe that a more precise, semantic characterization of effects arises from a relational perspective. Adopting this perspective, one can define the footprint of a command extensionally, using two unary properties and one binary property.

Capturing a command's write effect is easy with a unary property, 'writes c ws' stating that the initial and final heaps agree on the contents of their references, except those in ws.

```
type addrs = S.set addr
let writes (c:command) (ws:addrs) = ∀(h:heap).
 let h' = snd (reify (c ()) h) in
 (∀ r. r ∈ h ⟺ r ∈ h') ∧ (* no allocation *)
 (∀ r. addr_of r ∉ ws ⟹ h.[r] == h'.[r]) (* no changes except ws*)
```

Stating that a command only reads references rs is similar in spirit to noninterference (§6.5.1). Interestingly, it is impossible to describe the set of locations that a command may read without also speaking about the locations it may write. The relation 'reads c rs ws' states that if c writes at most the references in ws, then executing c in heaps that agree on the references in rs produces heaps that agree on ws, i.e., c does not depend on references outside rs.

```
let equiv_on (rs:addr_set) (h0:heap) (h1:heap) =
 ∀a (r:ref a). addr_of r ∈ rs ∧ r ∈ h0 ∧ r ∈ h1 ⟹ h0.[r] == h1.[r]
let reads (c:command) (rs ws:addrs) = ∀(h0 h1: heap).
 let h'0, h'1 = snd (reify (c ()) h0), snd (reify (c ()) h1) in
 (equiv_on rs h0 h1 ∧ writes c ws) ⟹ equiv_on ws h'0 h'1
```

Putting the pieces together, we define a read- and write-footprint-indexed type for commands:

```
type cmd (rs ws:addrs) = c:command{writes c ws ∧ reads c rs ws}
```

One can also define combinators to manipulate footprint-indexed commands. For example, here is a '»' combinator for sequential composition. Its type proves that read and write-footprints compose by a pointwise union, a higher-order relational property; the proof requires an (omitted) auxiliary lemma seq_lem (recall that variables preceded by a # are implicit arguments):

```
let seq (#r1 #w1 #r2 #w2 : addrs) (c1:cmd r1 w1) (c2:cmd r2 w2) :
   command = c1(); c2()
let (») #r1 #w1 #r2 #w2 (c1:cmd r1 w1) (c2:cmd r2 w2) :
   cmd (r1 ∪ r2) (w1 ∪ w2) = seq_lem c1 c2; seq c1 c2
```

Making use of relational footprints, we can prove other relations between commands, e.g., equivalences that justify program transformations. Command equivalence $c_0 \sim c_1$ states that running c_0 and c_1 in identical initial heaps produces (extensionally) equal final heaps.

```
let (∼) (c0:command) (c1:command) = ∀h.
  let h0, h1 = snd (reify (c0 ()) h), snd (reify (c1 ()) h) in
  ∀(r:ref α). (r ∈ h0 ⟺ r ∈ h1) ∧ (r ∈ h0 ⟹ h0.[r] == h1.[r])
```

For instance, we can prove that two commands can be swapped if they write to disjoint sets, and if the read footprint of one does not overlap with the write footprint of the other—this lemma is identical to a rule for swapping commands in a logic presented by [BGZ09].

```
let swap #rs1 #rs2 #ws1 #ws2 (c1:cmd rs1 ws1) (c2:cmd rs2 ws2)
   :Lemma (requires (disjoint ws1 ws2 ∧ disjoint rs1 ws2 ∧
                                disjoint rs2 ws1))
          (ensures ((c1 » c2) ∼ (c2 » c1)))
   = ∀_intro (λ h → let _ = reify (c1 ()) h, reify (c2 ()) h in
     () <: Lemma (equiv_on_h (c1 » c2) (c2 » c1) h))
```

The extended version [GMF+18] also verifies command idempotence and elimination of redundant writes.

6.3.2 Relational Hoare Logic

Beyond generic footprint-based transformations, one may also prove program-specific equivalences. Several logics have been devised for this, including, e.g., [Ben04] Relational Hoare logic (RHL). We show how to derive RHL within our framework by proving the soundness of each of its rules as lemmas about a program's reification.

Model To support potentially diverging computations, we instrument shallowly-embedded effectful computations with a *fuel* argument, where the value of the fuel is irrelevant for the behavior of a terminating computation.

```
type comp = f: (fuel:nat → St bool)
    { ∀h fuel fuel' . fst (reify (f fuel) h) == true ∧ fuel' > fuel
      ⟹ reify (f fuel') h == reify (f fuel) h }
let terminates_on c h = ∃fuel . fst (reify (c fuel) h) == true
```

We model effectful expressions whose evaluation always terminates and does not change the memory state, and assignments, conditionals, sequences of computations, and potentially diverging while loops.

Deriving RHL An RHL judgement 'related c_1 c_2 pre post' (where c_1, c_2 are effectful computations, and pre, post are relations over memory states) means that the executions of c_1, c_2 starting in memories h_1, h_2 related by pre, both diverge or both terminate with memories h_1', h_2' related by post.

```
let related (c1 c2 : comp) (pre post: (heap → heap → prop)) =
 (* if precondition holds on initial memory states, then *)
 ∀h1 h2 . pre h1 h2 ⟹
 (* c1 and c2 both terminate or both diverge, and *)
 ((c1 `terminates_on` h1 ⟺ c2 `terminates_on` h2) ∧
  (∀ fuel h1' h2' . (reify (c1 fuel) h1 == (true, h1') ∧
    reify (c2 fuel) h2 == (true, h2')) ⟹ (* if both terminate, *)
   post h1' h2')) (* postcondition holds on final memory states *)
```

From these reification-based definitions, we prove every rule of RHL. Of the 20 rules and equations of RHL presented by [Ben04], 16 need at most 5 lines of proof annotation each, among which 10 need none and are proven automatically. Rules related to while loops often require some manual induction on the fuel.

With RHL in hand, we can prove program equivalences applying syntax-directed rules, focusing the intellectual effort on finding and proving inductive invariants to relate loop bodies. When RHL is not powerful enough, we can escape back to the reification of commands to complete a direct proof in terms of the operational semantics. In the extended version [GMF+18] we sketch a program-specific equivalence built using our embedding of RHL in F*.

6.4 Cryptographic security proofs

We show how to construct a simple model for reasoning about probabilistic programs that sample values from discrete distributions. In this model, we prove the soundness of rules of probabilistic Relational Hoare Logic (pRHL) [BGZ09] allowing one to derive (in-)equalities on probability quantities from pRHL judgments. We illustrate our approach by formalizing a simple cryptographic proof: the perfect secrecy of one-time pad encryption .

The simplicity of our examples pales in comparison with complex proofs formalized in specialized tools based on pRHL like EasyCrypt [BGZ12] or FCF [PM15], yet our examples hint at a way to prototype and explore proofs in pRHL with a low entry cost.

6.4.1 A monad for random sampling

We begin by defining a monad for sampling from the uniform distribution over bitvectors of a fixed length q. We implement the monad as the composition of the state and exception monads where the state is a finite tape of bitvector values together with a pointer to a position in the tape. The RAND effect provides a single action, sample, which reads from the tape the value at the current position and advances the pointer to the next position, or raises an exception if the pointer is past the end of the tape.

```
type value = bv q
type tape = seq value
type id = i:ℕ{i < size}
type store = id * tape
```

```
type rand a = store → M (option a * id)
total new_effect {
  RAND: a:Type → Effect
  with repr = rand a;
       bind = λ(a b:Type) (c:rand a) (f:a → rand b) s →
                let r, next = c s in
                match r with
                | None → None, next
                | Some x → f x (next, snd s);
       return = λ(a:Type) (x:a) (next,_) → (Some x, next);
       sample = λ() s → let next, t = s in
                  if next + 1 < size then (Some (t n), n + 1)
                  else (None, n) }
effect Rand a = RAND a (λ initial_tape post → ∀x. post x)
```

Assuming a uniform distribution over initial tapes, we define the unnormalized measure of a function p:a $\to \mathbb{N}$ with respect to the denotation of a reified computation in f:Rand a as let mass f p = sum (λt $\to$ let r,_ = f (0, t) in p r) where sum: (tape $\to \mathbb{N}$) $\to \mathbb{N}$ is the summation operator over finite tapes. When p only takes values in $\{0, 1\}$, it can be regarded as an *event* whose probability with respect to the distribution generated by f is

$$\Pr[f : p] = \frac{1}{|\mathsf{tape}|} \times \sum_{t \in \mathsf{tape}} p\ (\mathsf{fst}\ (f\ t)) = \frac{\mathsf{mass}\ f\ p}{|\mathsf{tape}|}$$

We use the shorthand $\Pr[f = v] = |\mathsf{tape}|^{-1} \times \mathsf{mass}\ f\ (\mathsf{point}\ v)$ for the probability of a successful computation returning a value v, where let point x = λy $\to$ if y = Some x then 1 else 0.

6.4.2 Perfect secrecy of one-time pad encryption

The following effectful program uses a one-time key k sampled uniformly at random to encrypt a bitvector m:

```
let otp (m:value) : Rand value = let k = sample () in m ⊕ k
```

We show that this construction, known as *one-time pad*, provides *perfect secrecy*. That is, a ciphertext does not give away any information about the encrypted plaintext, provided the encryption key is used just once. Or equivalently, the distribution of the one-time pad encryption of a message is independent of the message itself, $\forall m_0,\ m_1,\ c.\ \Pr[\mathsf{otp}\ m_0 = c] = \Pr[\mathsf{otp}\ m_1 = c]$. We prove this by applying two rules of pRHL, namely [R-Rand] and [PrLe]. The former allows us to relate the results of two probabilistic programs by showing a bijection over initial random tapes that would make the relation hold (intuitively, permuting equally probable initial tapes does not change the resulting distribution over final tapes). The latter allows us to infer a probability inequality from a proven relation between probabilistic programs. Together, the two rules allow us to prove the following lemma:

$$\textsc{CSub}\ \frac{\Gamma, \mathrm{pc} : l_1 | - c \qquad l_2 \leq l_1}{\Gamma, \mathrm{pc} : l_2 | - c} \qquad \textsc{CAssign}\ \frac{\Gamma | - e : \Gamma(r)}{\Gamma, \mathrm{pc} : \Gamma(r) \,| - r := e}$$

$$\textsc{CCond}\ \frac{\Gamma | - e : l \qquad \Gamma, \mathrm{pc} : l | - c_1 \qquad \Gamma, \mathrm{pc} : l | - c_2}{\Gamma, \mathrm{pc} : l | - \text{if } e = 0 \text{ then } c_1 \text{ else } c_2}$$

Figure 6.1: A classic IFC type system (selected rules)

```
val mass_leq: #a:Type → #b:Type →
  c1:(store → M (a * id)) → c2:(store → M (b * id)) →
  p1:(a → nat) → p2:(b → nat) → bij:bijection → Lemma
  (requires (∀ t. let r1,_ = c1 (to_id 0,t) in
                  let r2,_ = c2 (to_id 0,bij.f t) in p1 r1 ≤ p2 r2))
  (ensures (mass c1 p1 ≤ mass c2 p2))
```

The proof is elementary from rearranging terms in summations according to the given bijection. The following secrecy proof of one-time pad is immediate from this lemma using as bijection on initial tapes $\lambda t \rightarrow$ upd t 0 (t 0 $\oplus$ m0 $\oplus$ m1):

```
val otp_secure: m0:value → m1:value → c:value → Lemma
  (let f0, f1 = reify (otp m0), reify (otp m1) in
   mass f0 (point c) == mass f1 (point c))
```

6.5 Information-flow control

In this section, we present a case study examining various styles of information-flow control (IFC), a security paradigm based on *noninterference* [GM82], a property that compares two runs of a program differing only in the program's secret inputs and requires the non-secret outputs to be equal. Many special-purpose systems, including syntax-directed type systems, have been devised to enforce noninterference-like security properties see e.g.,[SM06, HS12].

We start our IFC case study by encoding a classic IFC type system [VIS96] for a small deeply-embedded imperative language and proving its correctness (§6.5.1). In order to augment the permissiveness of our analysis we then show how to compose our IFC type system with precise semantic proofs (§6.5.2). In the extended version [GMF+18] we additionally treat a runtime monitor or IFC and delimited release. We conclude that our method for relational verification is flexible enough to accommodate various IFC disciplines, allowing comparisons and compositions within the same framework.

6.5.1 Deriving an IFC type system

Consider the following small *while* language consisting of expressions, which may only read from the heap, but not modify it, and commands, which may write to the heap and branch, depending on its contents. The definition of the language should be unsurprising, the only subtlety worth noting is the decr expression in the while command, a metric used to ensure loop termination.

$$\begin{array}{rcl} e & ::= & i \mid r \mid e_1 \oplus e_2 \\ c & ::= & \text{skip} \mid r := e \mid c_1; c_2 \mid \text{if } e = 0 \text{ then } c_1 \text{ else } c_2 \\ & & \mid \text{while } e \neq 0 \text{ do } c \,(\text{decr } e') \end{array}$$

A classic IFC type system [VIS96] devise an IFC type system to check that programs executing over a memory containing both secrets (stored in memory locations labeled High) and non-secrets (in locations labeled Low) never leak secrets into non-secret locations. The type system includes two judgments $\Gamma| - e : l$, which states that the expression e (with free variables in Γ) depends only on locations labeled l or lower; and $\Gamma, \text{pc} : l| - c$, which states that a command c in a context that is *control-dependent* on the contents of memory locations labeled l, does not leak secrets. Some selected rules of their system, as adapted to our example language, are shown in Figure 6.1.

Multiple effects to structure the *while* interpreter We deeply embed the syntax of *while* in F⋆ using data types exp and com, for expressions and commands, respectively. The expression interpreter interp_exp only requires reading the value of the variables from the store, whereas the command interpreter, interp_com, also requires writes to the store, where store is an integer store mapping a fixed set of integer references 'ref int' to int. Additionally, interp_com may also raise an Out_of_fuel exception when it detects that a loop may not terminate (e.g., because the claimed metric is not actually decreasing). We could define both interpreters using a single effect, but this would require us to prove that interp_exp does not change the store and does not raise exceptions. Avoiding the needless proof overhead, we use a Reader monad for interp_exp and StExn, a combined state and exceptions monad, for interp_com. By defining Reader as a sub_effect of StExn, expression interpretation is transparently lifted by F⋆ to the larger effect when interpreting commands. Using these effects, interp_exp and interp_com form a standard, recursive, definitional interpreter for *while*, with the following trivial signatures.

```
val interp_exp: exp → Reader int
val interp_com: com → StExn unit
```

Deriving IFC typing for expressions For starters, we use a store_labeling = ref int → label, where label ∈ {High, Low}, to partition the store between secrets (High) and non-secrets (Low). An expression is noninterferent at level l when its interpretation does not depend on locations labeled greater than l in the store. To formalize this, we define a notion of *low-equivalence* on stores, relating stores that agree on the contents of all Low-labeled references, and noninterferent expressions (at level Low, i.e., ni_exp env e Low) as those whose interpretation is identical in low-equivalent stores.

```
type low_equiv (env:store_labeling) (s0 s1:store) =
    ∀(r:ref int). env r=Low ⟹ s0.[r] == s1.[r]
let ni_exp (env:store_labeling) (e:exp) (l:label) =
    ∀(s0 s1:store). (low_equiv env s0 s1 ∧ l == Low) ⟹
      reify (interp_exp e) s0 == reify (interp_exp e) s1
```

With this definition of noninterference for expressions we capture the semantic interpretation of the typing judgment $\Gamma| - e : l$: if the expression e can be assigned the label Low, then the computation of e is only influenced by Low values.

Deriving IFC typing for commands As explained previously, the judgment $\Gamma, \text{pc} : l| - c$ deems c noninterferent when run in context control-dependent only on locations whose label is at most l. More explicitly, the judgment establishes the following two properties: (1) locations labeled below l are not modified by c—this is captured by no_write_down, a unary property; (2) the command c does not leak the contents of a High location to Low location—this is captured by ni_com', a binary property.

```
let run c s = match reify (interp_com c) s with
  | Inr Out_of_fuel, _ → Loops | _, s' → Returns s'
let no_write_down env c l s = match run c s with
  | Loops → ⊤| Returns s' → ∀(i:id). env i < l ⟹ s'.[i] == s.[i]
let ni_com' env c l s0 s1 = match run c s0, run c s1 with
  | Returns s0', Returns s1' → low_equiv env s0 s1 ⟹
      low_equiv env s0' s1'
  | Loops, _ | _, Loops → ⊤
```

The type system is termination-insensitive, meaning that a program may diverge depending on the value of a secret. Consider, for instance, two runs of the program `while hi <> 0 do {skip}; lo := 0`, one with `hi = 0` and another with `hi = 1`. The first run terminates and writes to `lo`; the second run loops forever. As such, we do not expect to prove noninterference in case the program loops. Putting the pieces together, we define $\Gamma, \text{pc} : l| - c$ to be ni_com Γ c l.

```
let ni_com (env:store_labeling) (c:com) (l:label) =
  (∀ s0 s1. ni_com' env c l s0 s1) ∧ (∀ s. no_write_down env c l s)
```

As in the case of expression typing, we derive each rule of the command-typing judgment as a lemma about ni_com. For example, here is the statement for the CCOND rule:

```
val cond_com (env:store_labeling)(e:exp)(ct:com)(cf:com)(l:label)
: Lemma (requires (ni_exp env e l ∧ ni_com env ct l
                                  ∧ ni_com env cf l))
        (ensures (ni_com env (If e ct cf) l))
```

The proofs of many of these rules are partially automated by SMT—they take about 250 lines of specification and proof in F*. Once proven, we use these rules to build a certified, syntax-directed

typechecker for *while* programs that repeatedly applies these lemmas to prove that a program satisfies ni_com. This typechecker has the following type:

```
val tc_com : env:store_labeling → c:com →
  Exn label (requires ⊤) (ensures λInl l → ni_com env c l l _ → ⊤)
```

6.5.2 Combining syntactic IFC analysis with semantic noninterference proofs

Building on §6.5.1, we show how programs that fall outside the syntactic information-flow typing discipline can be proven secure using a combination of typechecking and semantic proofs of noninterference. This example is evocative (though at a smaller scale) of the work of [KTB+15], who combine automated information-flow analysis in the Joana analyzer [HS09] with semantic proofs in the KeY verifier for Java programs [DHS05, SS11]. In contrast, we sketch a combination of syntactic and semantic proofs of relational properties in *a single* framework. Consider the following *while* program, where the label of c and lo is Low and the label of hi is High.

```
while c ≠0 do hi := lo + 1; lo := hi + 1; c := c − 1 (decr c)
```

The assignment lo := hi + 1 is ill-typed in the type system of §6.5.1, since it directly assigns a High expression to a Low location. However, the previous command overwrites hi so that hi does not contain a High value anymore at that point. As such, even though the IFC type system cannot prove it, the program is actually noninterferent. To prove it, one could directly attempt to prove ni_com for the entire program, which would require a strong enough (relational) invariant for the loop. A simpler approach is to prove just the sub-program hi := lo + 1; lo := hi + 1 (c_s) noninterferent, while relying on the type system for the rest of the program. The sub-program can be automatically proven secure:

```
let c_s_ni () : Lemma (ni_com env c_s Low) = ()
```

This lemma has exactly the form of the other standard, typing rules proven previously, except it is specialized to the command in question. As such, c_s_ni can just be used in place of the standard sequence-typing rule (CSEQ) when proving the while loop noninterferent.

We can even modify our automatic typechecker from §6.5.1 to take as input a list of commands that are already proved noninterferent (by whichever means), and simply look up the command it tries to typecheck in the list before trying to typecheck it syntactically. The type (and omitted implementation) of this typechecker is very similar to that of tc_com, the only difference is the extra list argument:

```
val tc_com_hybrid : env:store_labeling → c:com →
  list (cl:(com*label){ni_com env (fst cl) (snd cl)}) →
  Exn label (ensures λol → Inl? ol ⟹ ni_com env c (Inl?.v ol))
```

We can complete the noninterference proof automatically by passing the (c_s, Low) pair proved in ni_com by lemma c_s_ni (or directly by SMT) to this hybrid IFC typechecker:

```
let c_loop_ni () : Lemma (ensures ni_com env c_loop Low) =
 c_s_ni(); ignore (reify (tc_com_hybrid env c_loop [c_s, Low]) ())
```

Checking this in F* works by simply evaluating the invocation of tc_com_hybrid; this reduces fully to Inl Low and the intrinsic type of tc_com_hybrid ensures the postcondition.

6.6 Program optimizations and refinement

This section presents two complete examples to prove a few, classic algorithmic optimizations correct. These properties are very specific to their application domains and a special-purpose relational logic would probably not be suitable. Instead, we make use of the generality of our approach to prove application-specific relational properties (including 4- and 6-ary relations) of higher-order programs with local state. In contrast, most prior relational logics are specialized to proving binary relations, or, at best, properties of n runs of a single first-order program [SD16].

6.6.1 Effect for memoizing recursive functions

First, we look at memoizing total functions, including memoizing a function's recursive calls based on a partiality representation technique due to [McB15]. We prove that a memoized function is extensionally equal to the original.

We define a custom effect Memo, a monad with a state consisting of a (partial, finite) mapping from a function's domain type (dom) to its codomain type (codom), with two actions: get : dom → Memo (option codom), which returns a memoized value if such a value exists; and put : dom → codom → Memo unit, which adds a new memoization pair to the state.[3]

Take 1: Memoizing total functions Our goal is to turn a total function g into a memoized function f computing the same values as g. This relation between f's reification and g is captured by the computes predicate below, depending on an invariant of the memoization state, valid_memo. A memoization state h is valid for memoizing some total function g : (dom → codom) when h is a subset of the graph of g:

```
let valid_memo (h:memo_st) (g:dom → codom) =
  for_all_prop (λ (x,y) → y == g x) h
let computes (f: dom → Memo codom) (g:dom → codom) =
  ∀h0. valid_memo h0 g ⟹ (∀ x. (let y, h1 = reify (f x) h0 in
                                  y == g x ∧ valid_memo h1 g))
```

We have f `computes` g when given any state h0 containing a subgraph of g, f x returns g x and maintains the invariant that the result state h1 is a subgraph of g. It is easy to program and verify a simple memoizing function:

```
let memoize (g : dom → codom) (x:dom) =
  match get x with Some y → y | None → let y = g x in put x y; y
let memoize_computes g :Lemma ((memoize g) `computes` g) = ...
```

[3] This abstract model could be implemented efficiently, for instance by an imperative hash-table with a specific memory-management policy.

The proof of this lemma is straightforward: we only need to show that the value y we get back from the heap in the first branch is indeed g x which is enforced by the valid_memo in the precondition of computes.

Take 2: Memoizing recursive calls Now, what if we want to memoize a recursive function, for example, a function computing the Fibonacci sequence? We also want to memoize the intermediate recursive calls, and in order to achieve it, we need an explicit representation of the recursive structure of the function. Following [McB15], we represent this by a function x:dom → partial_result x, where a partial result is either a finished computation of type codom or a request for a recursive call together with a continuation.

```
type partial_result (x0:dom) =
  | Done : codom → partial_result x0
  | Need : x:dom{x ≺ x0} → cont:(codom → partial_result x0) →
           partial_result x0
```

As we define the fixed point using Need x f, we crucially require x ≺ x0, meaning that the value of the function is requested at a point x where function's definition already exists. For example encoding Fibonacci amounts to the following code where the two recursive calls in the second branch have been replaced by applications of the Need constructor. We also define the fixpoint of such a function representation f:

```
let fib_skel (x:dom) : partial_result x =
  if x ≤ 1 then Done 1 else
    Need (x − 1) (λ y1 → Need (x − 2) (λ y2 → Done (y1 + y2)))
let rec fixp (f: x:dom → partial_result x) (x0:dom) : codom =
  let rec complete_fixp x = function
    | Done y → y
    | Need x' cont → let y = fixp f x' in complete_fixp x (cont y)
  in complete_fixp x0 (f x0)
```

To obtain a memoized fixpoint, we need to memoize functions defined only on part of the domain, x:dom{p x}.

```
let partial_memoize (p:dom → Type)
  (f : x:dom{p x} → Memo codom) (x:dom{p x}) =
  match get x with Some y → y | None → let y = g x in put x y; y
let rec memoize_rec (f: x:dom → partial_result x) (x0:dom) =
  let rec complete_memo_rec x :Memo codom = function
    | Done y → y
    | Need x' cont →
      let y = partial_memoize (λ y → y ≺ x) (memoize_rec f) x' in
      complete_memo_rec (cont y)
  in complete_memo_rec x0 (f x0)
```

It is relatively easy to prove by structural induction on the code of memoize_rec that, for any skeleton of a recursive function f, we have that (memoize_rec f) `computes`(fixp f). The harder part is proving that fixp fib_skel is extensionally equal to fibonacci, the natural recursive definition of the sequence, as these two functions are not syntactically similar—however, the proof involves reasoning only about pure functions. As we have already proven that memoize_rec fib_skel computes fixp fib_skel, we easily gain a proof of the equivalence of memoize_rec fib_skel to fibonacci by transitivity.

6.6.2 Stepwise refinement and n-ary relations: Union-find with two optimizations

In this section, we prove several classic optimizations of a union-find data structure introduced in several stages, each a refinement. For each refinement step, we employ relational verification to prove that the refinement preserves the canonical structure of union-find. We specify correctness using, in some cases, 4- and 6-ary relations, which are easily manipulated in our monadic framework.

Basic union-find implementation A union-find data structure maintains disjoint partitions of a set, such that each element belongs to exactly one of the partitions. The data structure supports two operations: find, that identifies to which partition an element belongs, and union, that takes as input two elements and combines their partitions.

An efficient way to implement the union-find data structure is as a forest of disjoint trees, one tree for each partition, where each node maintains its parent and the root of each tree is the designated representative of its partition. The find operation returns the root of a given element's partition (by traversing the parent links), and the union operation simply points one of the roots to the other.

We represent a union-find of set $[0, \mathsf{n}-1]$ as the type 'uf_forest n' (below), a sequence of ref cells, where the i^{th} element in the sequence is the i^{th} set element, containing its parent and the list of all the nodes in the subtree rooted at that node. The list is computationally irrelevant (i.e., *erased*)—we only use it to express the disjointness invariant and the termination metric for recursive functions (e.g. find).

```
type elt (n:ℕ) = i:ℕ{i < n} × erased (list ℕ)
type uf_forest (n:ℕ) = s:seq (ref (elt n)){length s = n}
```

The basic find and union operations are shown below, where set and get are stateful functions that read and write the i^{th} index in the uf sequence. Reasoning about mutable pointer structures requires maintaining invariants regarding the liveness and separation of the memory referenced by the pointers. While important, these are orthogonal to the relational refinement proofs—so we elide them here, but still prove them intrinsically in our code.

```
let rec find #n uf i = let p, _ =
  get uf i in if p = i then i else find uf p
let union #n uf i₁ i₂ = let r₁, r₂ = find uf i₁, find uf i₂ in
```

```
let _, s1 = get uf r1 in let _, s2 = get uf r2 in
if r1 ≠r2 then (set uf r1 (r2, s1); set uf r2 (r2, union s1 s2))
```

Union by rank The first optimization we consider is union_by_rank, which decides whether to merge r_1 into r_2, or vice versa, depending on the heights of each tree, aiming to keep the trees shallow. We prove this optimization in two steps, first refining the representation of elements by adding a rank field to elt n and then proving that union_by_rank maintains the same set partitioning as union.

```
type elt (n:ℕ) = i:ℕ{i < n} × ℕ × erased (list nat) (* added rank *)
```

We formally reason about the refinement by proving that the outputs of the find and union functions do not depend on the newly added rank field. The rank_independence lemma (a 4-ary relation) states that find and union when run on two heaps that differ only on the rank field, output equal results and the resulting heaps also differ only on the rank field.

```
let equal_but_rank uf h1 h2 = ∀ i. parent uf i h1 = parent uf i h2
                                 ∧ subtree uf i h1 = subtree uf i h2
let rank_independence #n uf i i1 i2 h1 h2 : Lemma
(requires (equal_but_rank uf h1 h2))
(ensures (let (r1,f1), (r2,f2) =
   reify (find uf i) h1,reify (find uf i) h2 in
 let (_,u1), (_,u2) =
   reify (union uf i1 i2) h1,reify (union uf i1 i2) h2 in
 r1 == r2 ∧ equal_but_rank uf f1 f2 ∧ equal_but_rank uf u1 u2))
```

Next, we prove the union_by_rank refinement sound. Suppose we run union and union_by_rank in h on a heap h producing h_1 and h_2. Clearly, we cannot prove that find for a node j returns the same result in h_1 and h_2. But we prove that the canonical structure of the forest is the same in h_1 and h_2, by showing that two nodes are in the same partition in h_1 if and only if they are in the same partition in h_2:

```
val union_by_rank_refinement #n uf i1 i2 h j1 j2 : Lemma
  (let (_, h1), (_, h2) =
     reify (union uf i1 i2) h, reify (union_by_rank uf i1 i2) h in
   fst (reify (find uf j1) h1) == fst (reify (find uf j2) h1) ⟺
     fst (reify (find uf j1) h2) == fst (reify (find uf j2) h2))
```

This property is 6-ary relation, relating 1 run of union and 1 run of union_by_rank to 4 runs of find—its proof is a relatively straightforward case analysis.

Path compression Finally, we consider find_compress, which, in addition to returning the root for an element, sets the root as the element's new parent to accelerate subsequent find queries. To prove the refinement of find to find_compress sound, we prove a 4-ary relation showing that if running find and find_compress on a heap h results in the heaps h_1 and h_2, then the partition of a node j is the same in h_1 and h_2. This also implies that find_compress retains the canonical structure of the union-find forest.

```
val find_compress_refinement #n uf i h j
  : Lemma (let (r1, h1), (r2, h2) =
    reify (find uf i) h, reify (find_compress uf i) h in
    r1 == r2 ∧ fst (reify (find uf j) h1) == fst (reify (find uf j) h2))
```

6.7 Related work

Much of the prior related work focused on checking specific relational properties of programs, or general relational properties using special-purpose logics. In contrast, we argue that proof assistants that support reasoning about pure and effectful programs can, using our methodology, model and verify relational properties in a generic way. The specific incarnation of our methodology in F⋆ exploits its efficient implementation of effects enabled by abstraction and controlled reification; a unary weakest precondition calculus as a base for relational proofs; SMT-based automation; and the convenience of writing effectful code in direct style with returns, binds, and lifts automatically inserted.

Static IFC tools [SM03] survey a number of IFC type systems and static analyses for showing noninterference, trading completeness for automation. More recent verification techniques for IFC aim for better completeness [BH07, NBG13, AB04, ADZ+12, BNN16, SS11, BFG+14, Rab16], while compromising automation. The two approaches can be combined, as discussed in in §6.5.2.

Relational program logics and type systems A variety of program logics for reasoning about general relational properties have been proposed previously [Ben04, Yan07, BGZ09, ABG+17], while others apply general relational logics to specific domains, including access control [NBG13], cryptography [BGZ12, BGZ09, BDG+13, PM15], differential privacy [BKOZ13, ZK17], mechanism design [BGA+15], cost analysis [ÇBG+17], program approximations [CKMR12].

RF⋆, is worth pointing out for its connection to F⋆. [BFG+14] extend a prior, value-dependent version of F⋆ [SWS+13] with a probabilistic semantics and a type system that combines pRHL with refinement types. Like many other relational Hoare logics, RF⋆ provided an incomplete set of rules aimed at capturing many relational properties by intrinsic typing only.

In this paper we instead provide a versatile generic method for relational verification based on modeling effectful computations using monads and proving relational properties on their monadic representations, making the most of the support for full dependent types and SMT-based automation in the latest version of F⋆. This generic method can both be used directly to verify programs or as a base for encoding specialized relational program logics.

Product program constructions Product program constructions and self-composition are techniques aimed at reducing the verification of k-safety properties [CS10] to the verification of traditional (unary) safety proprieties of a product program that emulates the behavior of multiple input programs. Multiple such constructions have been proposed [BCK16] targeted for instance at secure IFC [TA05, BDR11, Nau06, YT14], program equivalence for compiler validation [ZP08], equivalence checking and computing semantic differences [LHKR12], program approximation [HLR16]. [SD16] recent Descartes tool for k-safety properties also creates k copies of the

program, but uses lockstep reasoning to improve performance by more tightly coupling the key invariants across the program copies. Recently [AGH+17] propose a tool that obtains better scalability by using a new decomposition of programs instead of using self-composition for k-safety problems.

Other program equivalence techniques Beyond the ones already mentioned above, many other techniques targeted at program equivalence have been proposed; we briefly review several recent works: [BKBH09] do manual proofs of correctness of compiler optimizations using partial equivalence relations. [KTL09] do automatic translation validation of compiler optimizations by checking equivalence of partially specified programs that can represent multiple concrete programs. [GS10] propose proof rules for proving the equivalence of recursive procedures. [LR15] and [cCLRR16] generalize this to a set of co-inductive equivalence proof rules that are language-independent. Automatically checking the equivalence of processes in a process calculus is an important building block for security protocol analysis [**?**, CCcCK16].

Semantic techniques Many semantic techniques have been proposed for reasoning about relational properties such as observational equivalence, including techniques based on binary logical relations [BKBH09, Mit86, ADR09, DNRB10, DAB11, DNB12, BHN13, BHN14], bisimulations [KW06, SKS11, Sum09] and combinations thereof [HDNV12, HNDV14]. While these very powerful techniques are often not directly automated, they can be used to provide semantic correctness proofs for relational program logics [DNRB10, DAB11] and other verification tools [BKHN16].

6.8 Conclusion

This paper advocates verifying relational properties of effectful programs using generic tools that are not specific to relational reasoning: monadic effects, reification, dependent types, non-relational weakest preconditions, and SMT-based automation. Our experiments in F* verifying relational properties about a variety of examples show the wide applicability of this approach. One of the strong points is the great flexibility in modelling effects and expressing relational properties about code using these effects. The other strong point is the good balance between interactive control, SMT-based automation, and the ability to encode even more automated specialized tools where needed. Thanks to this, the effort required from the F* programmer for relational verification seems on par with non-relational reasoning in F* and with specialized relational program logics.

CHAPTER 7

Conclusion and Directions for Future Research

7.1 Conclusion

In this thesis we have shown the importance and the practicality of static analysis for relational reasoning in security. We have developed novel verification frameworks, based on information flow control type systems, in the areas of cryptographic protocol analysis and web security.

We first proposed a two-layered approach of typing and constraint checking, that enforces observational equivalence for cryptographic protocols. Our evaluation has shown that this approach yields a highly efficient technique that is applicable to various protocols from the literature – including protocols that could previously not be analyzed by any automated tool.

We then instrumented a web browser, formalized as a reactive system, with a runtime monitor, parametrized by simple policies, and define a set of constraints for these policies that is sufficient to guarantee strong notions of confidentiality and integrity. We show how this approach can be employed to uniformly model existing browser side defense mechanisms and thus analyze their effectiveness.

We then developed a type system for web applications that enforces web session integrity in a core model of the web. We applied the approach to several real world examples, unveiling novel vulnerabilities and verifying the security of fixed versions.

Finally, we showed how the power of the semi-automated proof assistant F* can be used for the verification of relational properties for effectful program. We verify a variety of examples from different areas of research, including the correctness of an information flow control type system.

7.2 Directions for Future Research

In Chapter 6 we have shown how a proof assistant can be used for relational reasoning. While none of the examples presented there reaches the complexity of the other proofs presented in this thesis, large proofs have historically been done in proof assistants, and there is no obvious obstacle (besides the large investment of work time) that would prevent a formalization of the results of this thesis in a proof assistant. These formalizations would hence be an interesting future work, as it would further strengthen the confidence in the correctness of the results. Furthermore this could facilitate extending or changing parts of the system, as the proof assistant would highlight or in the best case even automatically cover the parts of the proof that need to be adapted.

The models in this thesis have been designed with the following considerations in mind: We want to faithfully model real world settings and have enough expressiveness to cover relevant case studies, while still allowing for detailed formal proofs. Nevertheless, all formal models presented in this thesis can be extended to be more expressive (e.g., by adding support for more cryptographic primitives) or to consider stronger attackers (e.g., by giving the attacker access to timing information). It would hence be interesting to perform extensions of the models in order to investigate the impact on the desired security policies and if necessary adapt the proof technique.

Formal verification is only effective, if there are no discrepancies between what has been analyzed and what is deployed in practice. However, the approaches presented in this thesis work on a higher-level formal model and we rely on a manual translation to/from real world code, leaving room for human error. It would be desirable to bridge this gap using a certified automated translation, to immediately carry over the correctness result to the real world system.

All the frameworks presented in this thesis rely on labelling of critical components. This is a task that currently needs to be performed manually and is thus an obstacle that could prevent the usage of the frameworks by a larger audience. It would hence be interesting to investigate to which extent labels can be inferred automatically. Some labels are crucial for the policy (i.e., which values should be kept secret) and have to be supplied by the developer or could potentially be guessed using heuristics, while other labels could be inferred automatically.

List of Figures

List of Tables

Bibliography

[008] Machine readable travel document. Technical Report 9303, International Civil Aviation Organization, 2008.

[AB04] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2004.

[Aba00] Martín Abadi. Security protocols and their properties. In *Foundations of Secure Computation*, volume for the 20th International Summer School on Foundations of Secure Computation held in Marktoberdorf Germany of *NATO Science Series*, pages 39–60. IOS Press, 2000.

[ABB+05] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In *17th International Conference on Computer Aided Verification, CAV'2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285, Edinburgh, Scotland, 2005. Springer.

[ABG+17] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. A relational logic for higher-order programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2017.

[ABL+10] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*, pages 290–304, 2010.

[ACC+08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-Based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, pages 1–10, 2008.

[ACC+13] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti. An Authentication Flaw in Browser-Based Single Sign-On Protocols: Impact and remediations. *Computers & Security*, 33:41–58, 2013.

[ACK16] Myrto Arapinis, Véronique Cortier, and Steve Kremer. When are three voters enough for privacy properties? In *21st European Symposium on Research in Computer Security (ESORICS'16)*, Lecture Notes in Computer Science, pages 241–260, Heraklion, Crete, 2016. Springer.

[ACRR09] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Untraceability in the applied pi calculus. In *1st International Workshop on RFID Security and Cryptography*, pages 1–6. IEEE, 2009.

[ACRR10] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *2nd IEEE Computer Security Foundations Symposium (CSF'10)*, pages 107–121. IEEE Computer Society Press, 2010.

[Adi08] Ben Adida. Helios: Web-based open-audit voting. In *17th USENIX Security symposium*, SS'08, pages 335–348. USENIX Association, 2008.

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 340–353. ACM, 2009.

[ADZ+12] Torben Amtoft, Josiah Dodds, Zhi Zhang, Andrew W. Appel, Lennart Beringer, John Hatcliff, Xinming Ou, and Andrew Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 369–389. Springer, 2012.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM, 2001.

[AF04] Martín Abadi and Cédric Fournet. Private authentication. *Theoretical Computer Science*, 322(3):427 – 476, 2004.

[AGH+17] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for k-safety. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017), to appear., 2017.

[AHKS16] Alejandro Aguirre, Cătălin Hriţcu, Chantal Keller, and Nikhil Swamy. From F* to SMT (extended abstract). Talk at 1st International Workshop on Hammers for Type Theories (HaTT), 2016.

[AHM+17] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 515–529. ACM, 2017.

[Ali14] Yasser Ali. Hacking paypal accounts with one click (patched), 2014. Available at `http://yasserali.com/hacking-paypal-accounts-with-one-click`.

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography. In *International Conference on Theoretical Computer Science (IFIP TCS2000)*, pages 3–22. Springer, 2000.

[ASK16] Kazuyuki Asada, Ryosuke Sato, and Naoki Kobayashi. Verifying relational properties of functional programs by first-order refinement. *Science of Computer Programming*, 2016.

[Atk09] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, 2009.

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, February–March 2008.

[Bar11] Adam Barth. Http state management mechanism, 2011. Available at `https://tools.ietf.org/html/rfc6265`.

[BBDM13] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *Proceedings of the 2nd International Conference on Principles of Security and Trust, POST 2013*, pages 126–146, 2013.

[BBDM14] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[BBF+11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8:1–8:45, 2011.

[BBG+17] Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Semslice: Exploiting relational verification for automatic program slicing. In *Integrated Formal Methods - 13th International Conference,*

IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings, volume 10510 of *Lecture Notes in Computer Science*, pages 312–319. Springer, 2017.

[BCEM11] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Resource-aware authorization policies for statically typed cryptographic protocols. In *24th IEEE Computer Security Foundations Symposium*, CSF '11, pages 83–98, Washington, DC, USA, 2011. IEEE Computer Society.

[BCEM13] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Logical foundations of secure resource management in protocol implementations. In *2nd International Conference on Principles of Security and Trust*, POST 2013, pages 105–125, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BCEM15] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Affine refinement types for secure distributed programming. *ACM Transactions on Programming Languages and Systems*, 37(4):11:1–11:66, 2015.

[BCF+14] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, Wilayat Khan, and Mauro Tempesta. Provably sound browser-based enforcement of web session integrity. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 366–380, 2014.

[BCF17] Michele Bugliesi, Stefano Calzavara, and Riccardo Focardi. Formal methods for web security. *Journal of Logic and Algebraic Programming*, 87:110–126, 2017.

[BCFK14] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Automatic and robust client-side protection for cookie-based sessions. In *Engineering Secure Software and Systems - 6th International Symposium, ESSoS 2014, Munich, Germany, February 26-28, 2014, Proceedings*, pages 161–178, 2014.

[BCFK15] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[BCJ+15] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in chromium. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[BCK16] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *J. Log. Algebr. Meth. Program.*, 85(5):847–859, 2016.

[BDG+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[BDH15] David Baelde, Stéphanie Delaune, and Lucca Hirschi. Partial order reduction for security protocols. In *26th International Conference on Concurrency Theory (CONCUR'15)*, volume 42 of *LIPIcs*, pages 497–510. Leibniz-Zentrum für Informatik, 2015.

[BDM13] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 653–670, 2013.

[BdNP02] Michele Boreale, Rocco de Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2002.

[BDR11] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[BDS15] David Basin, Jannik Dreier, and Ralf Sasse. Automated Symbolic Proofs of Observational Equivalence. In *22nd ACM SIGSAC Conference on Computer and Communications Security (ACM CCS 2015)*, pages 1144–1155. ACM, ACM, October 2015.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 14–25, New York, NY, USA, 2004. ACM.

[BFG+14] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. In *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, pages 193–206. ACM, 2014.

[BFM04] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Authenticity by tagging and typing. In *2004 ACM Workshop on Formal Methods in Security Engineering*, FMSE '04, pages 1–12, New York, NY, USA, 2004. ACM.

[BFM05] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Analysis of typed analyses of authentication protocols. In *18th IEEE Workshop on Computer Security Foundations*, CSFW '05, pages 112–125, Washington, DC, USA, 2005. IEEE Computer Society.

[BFM07] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[BGA+15] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. Higher-order approximate relational refinement types

for mechanism design and differential privacy. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 55–68. ACM, 2015.

[BGZ09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101, 2009.

[BGZ12] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Probabilistic relational Hoare logics for computer-aided security proofs. In *11th International Conference on Mathematics of Program Construction*, volume 7342 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2012.

[BH07] Lennart Beringer and Martin Hofmann. Secure information flow and program logics. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 233–248. IEEE Computer Society, 2007.

[BHM08] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 195–209, Washington, DC, USA, 2008. IEEE Computer Society.

[BHM14] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Union, Intersection and Refinement Types and Reasoning About Type Disjointness for Secure Protocol Implementations. *Journal of Computer Security*, 22:301–353, 2014.

[BHN13] Nick Benton, Martin Hofmann, and Vivek Nigam. Proof-relevant logical relations for name generation. In *11th International Conference on Typed Lambda Calculi and Applications*, volume 7941 of *Lecture Notes in Computer Science*, pages 48–60. Springer, 2013.

[BHN14] Nick Benton, Martin Hofmann, and Vivek Nigam. Abstract effects and proof-relevant logical relations. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 619–632. ACM, 2014.

[BJM08a] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 75–88, 2008.

[BJM08b] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 17–30, 2008.

[BKBH09] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations: higher-order store. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 301–312. ACM, 2009.

[BKHB06] Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. Reading, writing and relations. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, volume 4279 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2006.

[BKHN16] Nick Benton, Andrew Kennedy, Martin Hofmann, and Vivek Nigam. Counting successes: Effects and transformations for non-deterministic programs. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 56–72. Springer, 2016.

[BKOZ13] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3):9:1–9:49, 2013.

[BKU15] Bernhard Beckert, Vladimir Klebanov, and Mattias Ulbrich. Regression verification for java using a secure information flow calculus. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic, July 7, 2015*, pages 6:1–6:6. ACM, 2015.

[Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[Bla16] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, 2016.

[BLSS16] Musard Balliu, Benjamin Liebe, Daniel Schoepe, and Andrei Sabelfeld. JSLINQ: Building Secure Applications across Tiers. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy, CODASPY 2016*, pages 307–318, 2016.

[BMU08] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *IEEE Symposium on Security and Privacy*, SP '08, pages 202–215. IEEE Computer Society, 2008.

[BNN16] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. Relational logic with framing and hypotheses. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December*

13-15, 2016, Chennai, India, volume 65 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[BP10] Aaron Bohannon and Benjamin C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010.

[BPPR16] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. Cosmed: A confidentiality-verified social media platform. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.

[BPPR17] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 729–748. IEEE Computer Society, 2017.

[BPS+09] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 79–90, 2009.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, pages 409–426, 2006.

[BRGH14] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in webkit's javascript bytecode. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 159–178, 2014.

[ÇBG+17] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 316–329, 2017.

[CBR+11] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 227–238, 2011.

[CC08] Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 109–118, Alexandria, Virginia, USA, 2008. ACM Press.

[CCcCK16] Rohit Chadha, Vincent Cheval, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. *ACM Trans. Comput. Log.*, 17(4):23:1–23:32, 2016.

[CCD13] Vincent Cheval, Véronique Cortier, and Stéphanie Delaune. Deciding equivalence-based properties using constraint solving. *Theoretical Computer Science*, 492:1–39, 2013.

[CCK12] Rohit Chadha, Stefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Programming Languages and Systems —21th European Symposium on Programming (ESOP'12)*, volume 7211 of *Lecture Notes in Computer Science*, pages 108–127, Tallinn, Estonia, 2012. Springer.

[cCLRR16] Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Rosu. A language-independent proof system for full program equivalence. *Formal Asp. Comput.*, 28(3):469–497, 2016.

[CCP13] Vincent Cheval, Véronique Cortier, and Antoine Plet. Lengths may break privacy – or how to check for equivalences with length. In *25th International Conference on Computer Aided Verification (CAV'13)*, volume 8043 of *Lecture Notes in Computer Science*, pages 708–723, St Petersburg, Russia, 2013. Springer.

[CDD17] Véronique Cortier, Stéphanie Delaune, and Antoine Dallon. Sat-equiv: an efficient tool for equivalence properties. In *30th IEEE Computer Security Foundations Symposium (CSF'17)*. IEEE Computer Society Press, 2017.

[CEK+15] Véronique Cortier, Fabienne Eigner, Steve Kremer, Matteo Maffei, and Cyrille Wiedling. Type-based verification of electronic voting protocols. In *4th International Conference on Principles of Security and Trust - Volume 9036*, pages 303–323, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[CFG+20] Stefano Calzavara, Riccardo Focardi, Niklas Grimm, Matteo Maffei, and Mauro Tempesta. Language-based web session integrity. In *IEEE 33rd Computer Security Foundations Symposium, CSF 2020.*, pages 107–122. IEE, 2020.

[CFGM16a] Stefano Calzavara, Riccardo Focardi, Niklas Grimm, and Matteo Maffei. Micro-Policies for Web Session Security. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium, CSF 2016*, pages 179–193, 2016.

[CFGM16b] Stefano Calzavara, Riccardo Focardi, Niklas Grimm, and Matteo Maffei. Micro-policies for web session security, 2016. Available at `https://sites.google.com/site/micropolwebsese`.

[CFGT17] Veronique Cortier, Alicia Filipiak, Said Gharout, and Jacques Traore. Designing and proving an emv-compliant payment protocol for mobile devices. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, pages 467–480. IEEE Computer Society, 2017.

[CFST17] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the Web: A Journey into Web Session Security. *ACM Computing Surveys*, 50(1):13:1–13:34, 2017.

[CGLM17a] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A Type System for Privacy Properties. In *24th ACM Conference on Computer and Communications Security, CCS 2017*, pages 409–423. ACM, 2017.

[CGLM17b] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A type system for privacy properties (technical report). arXiv:1708.08340, 2017.

[CGLM17c] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Typeeq. Source Code, 2017. Available at `https://secpriv.tuwien.ac.at/tools/typeeq`.

[CGLM18a] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Equivalence properties by typing in cryptographic branching protocols. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10804 of *Lecture Notes in Computer Science*, pages 160–187. Springer, 2018.

[CGLM18b] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Equivalence Properties by Typing in Cryptographic Branching Protocols (Technical Report), 2018. Available at `https://sites.google.com/site/typesystemeq/`.

[Che14] Vincent Cheval. Apte: an algorithm for proving trace equivalence. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *Lecture Notes in Computer Science*, pages 587–592, Grenoble, France, 2014. Springer.

[Chl10] Adam Chlipala. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*, pages 105–118, 2010.

[CJ97] John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0, 1997.

[CKMR12] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 169–180. ACM, 2012.

[CMKW13] Alexei Czeskis, Alexander Moshchuk, Tadayoshi Kohno, and Helen J. Wang. Lightweight server support for browser-based CSRF protection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, pages 273–284, 2013.

[Cre08] Cas J. F. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[CRZ06] Véronique Cortier, Michaël Rusinowitch, and Eugen Zălinescu. *Relating Two Standard Notions of Secrecy*, pages 303–318. Springer Berlin Heidelberg, 2006.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CS11] Véronique Cortier and Ben Smyth. Attacking and fixing helios: An analysis of ballot secrecy. In *24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 297–311. IEEE Computer Society Press, 2011.

[CS13] Véronique Cortier and Ben Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 21(1):89–148, 2013.

[CSH09] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-Tier, Label-Based Security Enforcement for Web Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009*, pages 269–282, 2009.

[DAB11] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

[dADG+15] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 813–830, 2015.

[DHS05] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005.

[DKR09] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[DNB12] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012.

[DNRB10] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 185–198, 2010.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*, pages 109–124, 2010.

[DT10] Jeremy Dawson and Alwen Tiu. Automating open bisimulation checking for the spi calculus. In *23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 307–321. IEEE Computer Society, 2010.

[Ele15] Electronic Frontier Foundation. HTTPS Everywhere, 2015. Available at `https://www.eff.org/https-everywhere`.

[EM13] Fabienne Eigner and Matteo Maffei. Differential privacy by typing in security protocols. In *26th IEEE Computer Security Foundations Symposium*, CSF '13, pages 272–286, Washington, DC, USA, 2013. IEEE Computer Society.

[EMM06] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1–2):162–202, 2006.

[FC16] Stefan Fehrenbach and James Cheney. Language-integrated provenance. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 214–227. ACM, 2016.

[FHK19] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. An Extensive Formal Security Analysis of the OpenID Financial-Grade API. In *Proceedings of the 40th IEEE Symposium on Security and Privacy, S&P 2019*, pages 453–471, 2019.

[Fil94] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 446–457, New York, NY, USA, 1994. ACM.

[FKS16] Daniel Fett, Ralf Küsters, and Guido Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*, pages 1204–1215, 2016.

[FKS17] Daniel Fett, Ralf Küsters, and Guido Schmitz. The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium, CSF 2017*, pages 189–202, 2017.

[FM11] Riccardo Focardi and Matteo Maffei. *Types for Security Protocols*, pages 143–181. IOS Press, 2011.

[FM14] Daniele Filaretti and Sergio Maffeis. An Executable Formal Semantics of PHP. In *Proceedings of the 28th European Conference in Object-Oriented Programming, ECOOP 2014*, pages 567–592, 2014.

[GDNP12] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 748–759, 2012.

[GJ03] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–519, 2003.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. *1982 IEEE Symposium on Security and Privacy*, 00:11, 1982.

[GMF+18] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *The 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pages 130–145. ACM, 2018.

[GS10] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2010.

[HBS16] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. Information-flow security for javascript and its apis. *Journal of Computer Security*, 24(2):181–234, 2016.

[HDNV12] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and kripke logical relations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 59–72. ACM, 2012.

[HJB12] Jeff Hodges, Collin Jackson, and Adam Barth. Http strict transport security (hsts), 2012. Available at `https://tools.ietf.org/html/rfc6797`.

[HLR16] Shaobo He, Shuvendu K. Lahiri, and Zvonimir Rakamaric. Verifying relative safety, accuracy, and termination for program approximations. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, volume 9690 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2016.

[HNDV14] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. A logical step forward in parametric bisimulations. Technical Report MPI-SWS-2014-003, 2014.

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[HS12] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

[JBSP11] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 1531–1537, 2011.

[JKK06] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing Cross Site Request Forgery Attacks. In *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks, SecureComm 2006*, pages 1–10, 2006.

[JSH07] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 601–610, 2007.

[KCB+14] Wilayat Khan, Stefano Calzavara, Michele Bugliesi, Willem De Groef, and Frank Piessens. Client side web session integrity as a non-interference property. In *Information Systems Security - 10th International Conference, ICISS 2014, Hyderabad, India, December 16-20, 2014, Proceedings*, pages 89–108, 2014.

[KTB+15] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A hybrid approach for proving noninterference of Java programs. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 305–319. IEEE Computer Society, 2015.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 327–337. ACM, 2009.

[KW06] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 141–152. ACM, 2006.

[LHKR12] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In

Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, volume 7358 of *Lecture Notes in Computer Science*, pages 712–717. Springer, 2012.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[LR15] Dorel Lucanu and Vlad Rusu. Program equivalence by circular reasoning. *Formal Asp. Comput.*, 27(4):701–726, 2015.

[LV09] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 331–346, 2009.

[McB15] Conor McBride. Turing-completeness totally free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2015.

[Mit86] John C. Mitchell. Representation independence and data abstraction. In *POPL '86*, pages 263–276, New York, NY, USA, 1986. ACM.

[MIT18a] MITRE. CVE-2018-10188, 2018.

[MIT18b] MITRE. CVE-2018-16854, 2018.

[MIT18c] MITRE. CVE-2018-19969, 2018.

[MIT19] MITRE. CVE-2019-12616, 2019.

[MMB+13] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 415–429. IEEE Computer Society, 2013.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.

[Moo] Moodle HQ. Moodle Learning Platform.

[MPR13] Matteo Maffei, Kim Pecina, and Manuel Reinert. Security and privacy by declarative design. In *26th IEEE Computer Security Foundations Symposium*, CSF '13, pages 81–96, Washington, DC, USA, 2013. IEEE Computer Society.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[MT09] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 77–91, 2009.

[Nau06] David A. Naumann. From coupling relations to mated invariants for checking information flow. In *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2006.

[NBG13] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 35(2):6, 2013.

[NMB08] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.

[NMY+11] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. Sessionshield: Lightweight protection against session hijacking. In *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, pages 87–100, 2011.

[Nog02] Aleksey Nogin. Quotient types: A modular approach. In *15th International Conference on Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 2002.

[OWvOS08] Terri Oda, Glenn Wurster, Paul C. van Oorschot, and Anil Somayaji. SOMA: mutual approval for included content in web pages. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 89–98, 2008.

[Pey10] Simon Peyton Jones. *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, 2010.

[php] phpMyAdmin Development Team. phpMyAdmin Database Administration Software.

[PM15] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9036 of *Lecture Notes in Computer Science*, pages 53–72. Springer, 2015.

[Rab16] Markus N. Rabe. *A temporal logic approach to Information-flow control*. PhD thesis, Saarland University, 2016.

[RBGH15] Vineet Rajani, Abhishek Bichhawat, Deepak Garg, and Christian Hammer. Information flow control for event handling and the DOM in web browsers. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 366–379, 2015.

[RDJP11] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and precise client-side protection against CSRF attacks. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, pages 100–116, 2011.

[RKW12] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 155–168, 2012.

[Roe16] Peter Roenne. Private communication, 2016.

[SD16] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 57–69. ACM, 2016.

[SEMM14] Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. A Formal Definition of Protocol Indistinguishability and Its Verification Using Maude-NPA. In *STM 2014*, Lecture Notes in Computer Science, pages 162–177. IEEE Computer Society, 2014.

[SGLH11] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, ICFP '11, pages 15–27, 2011.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016.

[SHS14] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. SeLINQ: Tracking Information Across Application-Database Boundaries. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014*, pages 25–38, 2014.

[SKS11] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SM06] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, 2006.

[SMCB12] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *24th IEEE Computer Security Foundations Symposium (CSF'12)*, pages 78–94. IEEE Computer Society, 2012.

[SS11] Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of Java programs without approximations. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*, volume 7421 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 2011.

[SSM10] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 921–930, 2010.

[Sum09] Eijiro Sumii. A complete characterization of observational equivalence in polymorphic *lambda*-calculus with general references. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 2009.

[SWS+13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[SYM+14] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, David Herman, Brad Karp, and David Mazières. Protecting users by confining javascript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 131–146, 2014.

[TA05] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[TDK11] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 615–626, 2011.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.

[W3C16] W3C. Content Security Policy Level 2, 2016.

[WBS11] Joel Weinberger, Adam Barth, and Dawn Song. Towards client-side HTML security policies. In *6th USENIX Workshop on Hot Topics in Security, HotSec'11, San Francisco, CA, USA, August 9, 2011*, 2011.

[WBV15] Mike West, Adam Barth, and Dan Veditz. Content security policy (csp), 2015. Available at `http://www.w3.org/TR/CSP/`.

[Wes] Mike West. Cookie Prefixes.

[Yan07] Hongseok Yang. Relational separation logic. *Theorertical Compututer Science*, 375(1-3):308–334, 2007.

[YNKM09] Alexander Yip, Neha Narula, Maxwell N. Krohn, and Robert Morris. Privacy-preserving browser-side scripting with bflow. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 233–246, 2009.

[YT14] Hirotoshi Yasuoka and Tachio Terauchi. Quantitative information flow as safety and liveness hyperproperties. *Theor. Comput. Sci.*, 538:167–182, 2014.

[ZJL+15] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies lack integrity: Real-world implications. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 707–721, 2015.

[ZK17] Danfeng Zhang and Daniel Kifer. LightDP: towards automating differential privacy proofs. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 888–901. ACM, 2017.

[ZP08] Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2008.

APPENDIX A

Appendix to Chapter 4

A.1 Additional Formal Details

A.1.1 Formal Definition of the Reactive Semantics

Before presenting the semantics, we first introduce a little bit of additional notation. We write *transfer*(event_type, τ_1, τ_2) $\downarrow$ when *transfer* is defined for an event of type event_type and a pair of tags τ_1, τ_2, otherwise we write *transfer*(event_type, τ_1, τ_2) $\uparrow$. Notice that the notation *transfer*(event_type, τ_1, τ_2) $\downarrow$ is just a shorthand for *transfer*(event_type, τ_1, τ_2) $= (-, -, -, -)$.

The function $!K(u)^{\tau_r}$ retrieves from the cookie jar K all the cookies which should be read by the origin u, based on the tag τ_r passed to the cookie jar. Intuitively, the tag τ_r models the security assumptions on the reader: when cookies are fetched by the browser for inclusion in a HTTP(S) request to u, it may simply be the protocol of u; when cookies are accessed by a script running in the origin u, it may reflect the amount of trust we place on the script. The tag τ_r is indirectly passed to the transfer function in the definition of $!K(u)^{\tau_r}$ for an event of type get: only cookies set by the domain of u with a tag τ such that *transfer*(get, τ_r, τ) is defined are returned. Formally, we have:

$$!K(u)^{\tau_r} = \{\mathsf{ck}(k, v)^{\tau} \in K(\mathit{host}(u)) \mid \mathit{transfer}(\mathsf{get}, \tau_r, \tau)\downarrow\}$$

We let $K \xleftarrow{CK}_{\tau_w} u$ be the cookie jar obtained as the result of updating the cookie jar K with the cookies CK set by the origin u; here, the tag τ_w represents the security assumptions on the writer. Again, this tag may correspond to the protocol of u for cookies set via HTTP(S) responses from u or it may reflect the trust we place in a script setting cookies while running in the origin u. Formally, we let $K \xleftarrow{CK}_{\tau_w} u$ be the map coinciding with K, but for the entry $d = \mathit{host}(u)$, defined as follows:

$$\{\mathsf{ck}(k, v)^{\tau} \in CK \mid \mathit{transfer}(\mathsf{set}, \tau_w, \tau)\downarrow\} \cup$$
$$\{\mathsf{ck}(k, v)^{\tau} \in K(d) \mid \forall\, \mathsf{ck}(k', v')^{\tau} \in CK : k' \neq k \vee \mathit{transfer}(\mathsf{set}, \tau_w, \tau)\uparrow\}$$

The definition includes two cases. In words, it states that, after the update, the cookie jar includes: (1) all the cookies in CK which can be legitimately set based on the transfer function for an event of type set, and (2) all the cookies in K which are not overwritten by cookies in CK, either because no cookie with the same key is included in CK or because the transfer function is undefined.

The transitions $C \xrightarrow{i} P$ in Table A.1 describe how the consumer state C reacts to the input event i by evolving into a producer state P. The definition of $C \xrightarrow{i} P$ consists only of two rules: (I-Mirror) and (I-Complete), which are needed to make FF^{τ} a reactive system. The bulk of the semantics is defined by the auxiliary relation $C \overset{i}{\mapsto} P$.

(I-LOAD)
$$\frac{\mathit{transfer}(\mathsf{load}, \mathit{tag}(u), -) = (\tau_n, -, \tau_{co}, -)}{\langle K, N, H, \mathsf{wait}, [\,]\rangle \xrightarrow{\mathsf{load}(u)} \langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, \mathsf{doc\_req}(u : !K(u)^{\tau_{co}})\rangle}$$

(I-DOCRESP)
$$\frac{\mathit{transfer}(\mathsf{doc\_resp}, \tau_n, -) = (-, \tau_{ci}, -, \tau_s)}{\langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, [\,]\rangle \xrightarrow{\mathsf{doc\_resp}_n(u:CK,e)} \langle K \overset{CK}{\leftarrow}_{\tau_{ci}} u, N, H, \lceil e \rfloor^{\tau_s}_{@u}, [\,]\rangle}$$

(I-DOCREDIR)
$$\frac{\mathit{transfer}(\mathsf{doc\_redir}, \tau_n, \mathit{tag}(u')) = (\tau_m, \tau_{ci}, \tau_{co}, -) \qquad K' = K \overset{CK}{\leftarrow}_{\tau_{ci}} u \qquad N' = N \uplus \{n^{\tau_m} : u'\}}{\langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, [\,]\rangle \xrightarrow{\mathsf{doc\_redir}_n(u:CK,u')} \langle K', N', H, \mathsf{wait}, \mathsf{doc\_req}(u' : !K'(u)^{\tau_{co}})\rangle}$$

(I-XHRRESP)
$$\frac{\mathit{transfer}(\mathsf{xhr\_resp}, \tau_n, -) = (-, \tau_{ci}, -, \tau_s)}{\langle K, N, H \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u'})\}, \mathsf{wait}, [\,]\rangle \xrightarrow{\mathsf{xhr\_resp}_n(u:CK,v)} \langle K \overset{CK}{\leftarrow}_{\tau_{ci}} u, N, H, \lceil e\{v/x\} \rfloor^{\tau_s}_{@u'}, [\,]\rangle}$$

(I-XHRREDIR)
$$\frac{\mathit{transfer}(\mathsf{xhr\_redir}, \tau_n, \mathit{tag}(u')) = (\tau_m, \tau_{ci}, \tau_{co}, -) \qquad K' = K \overset{CK}{\leftarrow}_{\tau_{ci}} u}{\begin{array}{c}\langle K, N, H \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u''})\}, \mathsf{wait}, [\,]\rangle \xrightarrow{\mathsf{xhr\_redir}_n(u:CK,u')} \\ \langle K', N, H \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K'(u')^{\tau_{co}})\rangle\end{array}}$$

(I-MIRROR)
$$\frac{C \overset{i}{\mapsto} P}{C \xrightarrow{i} P}$$

(I-COMPLETE)
$$\frac{\nexists P : \langle K, N, H, \mathsf{wait}, [\,]\rangle \overset{i}{\mapsto} P}{\langle K, N, H, \mathsf{wait}, [\,]\rangle \xrightarrow{i} \langle K, N, H, \mathsf{wait}, \bullet\rangle}$$

Table A.1: Reactive semantics of FF^{τ} - Input events

When the user navigates the browser to a URL u, a new network connection n is created and is assigned a new tag τ_n by rule (I-Load), based on the tag of the URL u. Also, a new tag τ_{co} is computed and passed to the cookie jar, to determine the set of cookies to be sent to u, and a new document request event including these cookies is put into the output buffer. If a HTTP(S)

redirect to a URL u' is received over n, the tag of the network connection may be updated to a new tag τ_m as the result of rule (I-DocRedir). The update is determined by the tag τ_n of the network connection and the tag of the redirect URL u'. Notice that rule (I-DocRedir) also computes two tags which are needed to set and get cookies upon redirection respectively. When a document response is eventually received over a network connection, the connection is closed and a new script is run in the browser by rule (I-DocResp). The tag assigned to the script is computed by rule (I-DocResp), based on the tag of the network connection where the script is received. Observe that the tag of the network connection may actually keep track of the entire redirection chain which is followed before the script is downloaded. The treatment for XHR requests and responses is analogous to what we discussed above. The only difference is that XHR connections are not opened as the result of processing a $\mathsf{load}(u)$ event, but rather by script execution, which is explained in the next sub-section.

The transitions $P \xrightarrow{o} Q$ in Table A.2 describe how a producer state P generates an output event o and evolves into another state Q. Like for inputs, the semantics is split in two layers, with the rules (O-Mirror) and (O-Complete) accommodating the requirements of the reactive system definition and an auxiliary relation $P \overset{o}{\mapsto} Q$ formalizing the non-trivial moves. The first three rules for this relation are standard, so we just comment on the remaining rules.

Rules (O-GetCookie) and (O-SetCookie) model cookie access and cookie setting via JavaScript respectively. These rules make use of the function for getting and setting cookies we already discussed. The only point worth mentioning here is that the outcome of these two operations is determined by both the tag assigned to the script and the tag assigned to the cookie which is read/set. Notice that if a cookie with key k is set by a script running in the origin u, it always gets the tag $\kappa(\mathit{host}(u), k)$ intended for cookies with key k set by the domain owning the script. Hence, existing JavaScript code does not need to be adapted to include the tags intended for the cookies it sets.

Rule (O-Xhr) models the sending of an AJAX request by a script. Based on the tag of the script τ_s and the tag of the destination URL u', new tags τ_n and τ_{co} are computed for the new network connection and for accessing the cookie jar to fetch the cookies to be attached to the request. When the AJAX request is sent, its asynchronous behaviour is captured by binding the continuation $\lambda x.e$ to the new network connection where a response is expected and by immediately returning the dummy value unit, so that the script can immediately proceed in its execution. The continuation $\lambda x.e$ is later executed in the origin where the request was sent as the result of an application of rule (I-XhrResp), processing the corresponding response.

Finally, rule (O-Flush) is used to flush the output buffer when it is populated by an application of rule (I-Load): this is technically needed to move from a producer state back into a consumer state.

A.1.2 Constraint System

We propose a set of constraints on the transfer function, which ensure that a number of useful non-interference properties are enforced at runtime by FF^τ. Specifically, we are able to support:

(O-APP)
$$\langle K, N, H, \lceil (\lambda x.e)\ v \rfloor^{\tau_s}_{@u}, [] \rangle \overset{\bullet}{\mapsto} \langle K, N, H, \lceil e\{v/x\} \rfloor^{\tau_s}_{@u}, [] \rangle$$

(O-LETCTX)
$$\frac{\langle K, N, H, \lceil e \rfloor^{\tau_s}_{@u}, [] \rangle \overset{o}{\mapsto} \langle K', N', H', \lceil e'' \rfloor^{\tau_s}_{@u}, [] \rangle}{\langle K, N, H, \lceil \mathsf{let}\ x = e\ \mathsf{in}\ e' \rfloor^{\tau_s}_{@u}, [] \rangle \overset{o}{\mapsto} \langle K', N', H', \lceil \mathsf{let}\ x = e''\ \mathsf{in}\ e' \rfloor^{\tau_s}_{@u}, [] \rangle}$$

(O-LET)
$$\langle K, N, H, \lceil \mathsf{let}\ x = v\ \mathsf{in}\ e \rfloor^{\tau_s}_{@u}, [] \rangle \overset{\bullet}{\mapsto} \langle K, N, H, \lceil e\{v/x\} \rfloor^{\tau_s}_{@u}, [] \rangle$$

(O-GETCOOKIE)
$$\frac{\exists \tau, v : \mathsf{ck}(k, v)^{\tau} \in\ !K(u)^{\tau_s}}{\langle K, N, H, \lceil \mathsf{get\text{-}ck}(k) \rfloor^{\tau_s}_{@u}, [] \rangle \overset{\bullet}{\mapsto} \langle K, N, H, \lceil v \rfloor^{\tau_s}_{@u}, [] \rangle}$$

(O-SETCOOKIE)
$$\frac{CK = \{\mathsf{ck}(k, v)^{\tau} \mid \tau = \kappa(\mathit{host}(u), k)\}}{\langle K, N, H, \lceil \mathsf{set\text{-}ck}(k, v) \rfloor^{\tau_s}_{@u}, [] \rangle \overset{\bullet}{\mapsto} \langle K \overset{CK}{\longleftarrow}_{\tau_s} u, N, H, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [] \rangle}$$

(O-XHR)
$$\frac{\mathit{transfer}(\mathsf{send}, \tau_s, \mathit{tag}(u')) = (\tau_n, -, \tau_{co}, -)}{\langle K, N, H, \lceil \mathsf{xhr}(u', \lambda x.e) \rfloor^{\tau_s}_{@u}, [] \rangle \xrightarrow{\mathsf{xhr\_req}(u' : !K(u')^{\tau_{co}})} \langle K, N, H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [] \rangle}$$

(O-FLUSH)
$$\langle K, N, H, T, o :: O \rangle \overset{o}{\mapsto} \langle K, N, H, T, O \rangle$$

(O-MIRROR)
$$\frac{P \overset{o}{\mapsto} Q}{P \overset{o}{\to} Q}$$

(O-COMPLETE)
$$\frac{\nexists o, Q : \langle K, N, H, \lceil e \rfloor^{\tau_s}_{@u}, [] \rangle \overset{o}{\mapsto} Q}{\langle K, N, H, \lceil e \rfloor^{\tau_s}_{@u}, [] \rangle \overset{\bullet}{\to} \langle K, N, H, \mathsf{wait}, [] \rangle}$$

Table A.2: Reactive semantics of FF^{τ} - Output events

1. confidentiality of cookies: the actual value of a high-confidentiality cookie has no visible import for the attacker (no cookie theft);

2. integrity of cookies: a high-integrity cookie cannot be set or modified by the attacker (no session fixation);

3. confidentiality of web sessions: the attacker is unaware of the presence of any on-going session between the browser and a trusted website;

4. integrity of web sessions: the attacker cannot force the browser into establishing new sessions with a trusted website (no login CSRF), or into introducing additional messages into existing sessions (no CSRF).

Constraint-checks are formulated as follows:

$$\Gamma \vdash \textit{transfer}(\mathsf{event\_type}, \tau_1, \tau_2) = (\ell_n, \ell_{ci}, \ell_{co}, \ell_s),$$

reading as: the entry of the transfer function fulfills the constraints in an environment Γ. We write $\Gamma \vdash \textit{transfer}$ when each entry of the transfer function fulfills the constraints. If the transfer function fulfills the constraints, it provably enforces meaningful non-interference policies on cookies and network communication. Specifically, it ensures that:

1. if a cookie is given label ℓ, its value can only be disclosed to an attacker ℓ' such that $C(\ell) \cap C(\ell') \neq \emptyset$ (*cookie confidentiality*);
2. if a cookie is given label ℓ, it can only be set or modified by an attacker ℓ' such that $I(\ell) \cap I(\ell') \neq \emptyset$ (*cookie integrity*);
3. if a URL u is given label ℓ, an attacker ℓ' can notice that the browser is loading u only if $C(\ell) \cap C(\ell') \neq \emptyset$ (*session confidentiality*);
4. if a URL u is given label ℓ, an attacker ℓ' can force the browser to send requests to u only if $I(\ell) \cap I(\ell') \neq \emptyset$ (*session integrity*)

These properties are enforced by assigning labels to different browser elements, e.g., network connections and scripts, and by ensuring that these labels constitute upper bounds for the security relevant side-effects which can be triggered by these browser elements.

The constraints are given in Table A.3, we comment on them in the following. Before diving into the details, however, it is worth discussing how the functions *evt_label* and *msg_label* are consistently used in the rules: this is not obvious, since the attacker capabilities discriminate between message presence (*evt_label*) and message contents (*msg_label*). Intuitively, in the constraint system we only use *msg_label* when determining the set of cookies to be attached to a HTTP(S) request, since we only care about whether the remote end-point of a network connection should be entitled to get some cookies; in all the other cases, we use *evt_label* to ensure the absence of implicit information flows based on message presence. Recall that $\textit{msg\_label}(u) \subseteq C(\ell)$ implies $\textit{evt\_label}(u) \subseteq C(\ell)$ for all URLs u and all the well-formed attackers ℓ.

Rule (T-Load) ensures that, when a URL u is loaded, the information $\Gamma_C(u)$ is an upper bound for both $\textit{evt\_label}(u)$ and the confidentiality label $C(\ell_n)$ of the new network connection. Having $\textit{evt\_label}(u) \subseteq \Gamma_C(u)$ implies that the load event is always visible to any network attacker or any web attacker sitting at $\textit{host}(u)$, while having $C(\ell_n) \subseteq \Gamma_C(u)$ guarantees that the side-effects produced by a response received over the network connection are only visible to $\Gamma_C(u)$. The rule also checks two other conditions: $\textit{msg\_label}(u) \subseteq C(\ell_{co})$ is needed to ensure that the cookies attached to the document request sent to u can actually be disclosed to it, while $\textit{msg\_label}(u) \subseteq I(\ell_n)$ formalizes that an attacker who controls u may be able to compromise the integrity of any response received over the new network connection.

(T-LOAD)
$$\frac{\begin{array}{c}\mathit{evt\_label}(u) \cup C(\ell_n) \subseteq \Gamma_C(u) \\ \mathit{msg\_label}(u) \subseteq C(\ell_{co}) \qquad \mathit{msg\_label}(u) \subseteq I(\ell_n)\end{array}}{\Gamma \vdash \mathit{transfer}(\mathsf{load}, u, -) = (\ell_n, -, \ell_{co}, -)}$$

(T-DOCRESP)
$$\frac{\begin{array}{c}C(\ell_{ci}) \cup C(\ell_s) \subseteq C(\ell_n) \\ I(\ell_n) \subseteq I(\ell_{ci}) \cap I(\ell_s)\end{array}}{\Gamma \vdash \mathit{transfer}(\mathsf{doc\_resp}, \ell_n, -) = (-, \ell_{ci}, -, \ell_s)}$$

(T-DOCREDIR)
$$\frac{\begin{array}{c}\mathit{evt\_label}(u) \cup C(\ell_{ci}) \cup C(\ell_m) \subseteq C(\ell_n) \qquad \mathit{msg\_label}(u) \subseteq C(\ell_{co}) \\ I(\ell_n) \cup \mathit{msg\_label}(u) \subseteq I(\ell_m) \qquad I(\ell_n) \subseteq I(\ell_{ci}) \cap \Gamma_I(u)\end{array}}{\Gamma \vdash \mathit{transfer}(\mathsf{doc\_redir}, \ell_n, u) = (\ell_m, \ell_{ci}, \ell_{co}, -)}$$

(T-XHRRESP)
$$\frac{\begin{array}{c}C(\ell_{ci}) \cup C(\ell_s) \subseteq C(\ell_n) \\ I(\ell_n) \subseteq I(\ell_{ci}) \cap I(\ell_s)\end{array}}{\Gamma \vdash \mathit{transfer}(\mathsf{xhr\_resp}, \ell_n, -) = (-, \ell_{ci}, -, \ell_s)}$$

(T-XHRREDIR)
$$\frac{\begin{array}{c}\mathit{evt\_label}(u) \cup C(\ell_{ci}) \cup C(\ell_m) \subseteq C(\ell_n) \qquad \mathit{msg\_label}(u) \subseteq C(\ell_{co}) \\ I(\ell_n) \cup \mathit{msg\_label}(u) \subseteq I(\ell_m) \qquad I(\ell_n) \subseteq I(\ell_{ci}) \cap \Gamma_I(u)\end{array}}{\Gamma \vdash \mathit{transfer}(\mathsf{xhr\_redir}, \ell_n, u) = (\ell_m, \ell_{ci}, \ell_{co}, -)}$$

(T-GET)
$$\frac{C(\ell_r) \subseteq C(\ell_t) \qquad I(\ell_t) \subseteq I(\ell_r)}{\Gamma \vdash \mathit{transfer}(\mathsf{get}, \ell_r, \ell_t) = (-, -, -, -)}$$

(T-SET)
$$\frac{C(\ell_t) \subseteq C(\ell_w) \qquad I(\ell_w) \subseteq I(\ell_t)}{\Gamma \vdash \mathit{transfer}(\mathsf{set}, \ell_w, \ell_t) = (-, -, -, -)}$$

(T-SEND)
$$\frac{\begin{array}{c}\mathit{evt\_label}(u) \cup C(\ell_n) \subseteq C(\ell_s) \qquad \mathit{msg\_label}(u) \subseteq C(\ell_{co}) \\ I(\ell_s) \cup \mathit{msg\_label}(u) \subseteq I(\ell_n) \qquad I(\ell_s) \subseteq \Gamma_I(u)\end{array}}{\Gamma \vdash \mathit{transfer}(\mathsf{send}, \ell_s, u) = (\ell_n, -, \ell_{co}, -)}$$

Table A.3: A non-interference constraint system for transfer functions

Rules (T-DocResp) and (T-XhrResp) check that the confidentiality label $C(\ell_n)$ of the network connection where a response is received is an upper bound for the confidentiality label $C(\ell_{ci})$ of the cookies which are set in the HTTP(S) headers of the response, so that the attacker cannot infer the occurrence of private input events from the value of public cookies possibly set in such events. Also, the rules check that $C(\ell_n)$ is an upper bound for the confidentiality label $C(\ell_s)$ of the script downloaded over the network connection, so that the script cannot produce unexpected visible

side-effects. The conditions on integrity are dual: a network connection can only be used to set cookies and spawn scripts with lower integrity than the integrity label of the connection itself.

Rules (T-DocRedir) and (T-XhrRedir) are more complex, since HTTP(S) redirects have a number of side-effects: a new network connection is opened, new cookies are set into the browser, and cookies available in the cookie jar are fetched to compose an HTTP(S) request. Enforcing non-interference in presence of a redirect to u thus requires some care. The confidentiality label $C(\ell_n)$ of the network connection where the redirect is received must be an upper bound for: (1) *evt_label*(u), since a network request is sent to u and made visible to any attacker who controls the URL; (2) the label $C(\ell_{ci})$ of the new cookies set in the browser, otherwise these cookies could be exploited to reveal the occurrence of a redirect over a high-confidentiality connection, and (3) the label $C(\ell_m)$ of the new network connection, which otherwise may leak the existence of a previously opened connection. To ensure the confidentiality of the cookies sent in the redirect, the condition *msg_label*$(u) \subseteq C(\ell_{co})$ must be checked, just like in the case of rule (T-Load). As to integrity, two conditions must be verified. First, the integrity label $I(\ell_m)$ of the new network connection must be an upper bound for the integrity label of the original connection $I(\ell_n)$ and for *msg_label*(u), so that low-integrity connections cannot ever be endorsed to high integrity and all the origins involved in a redirection chain are actually tracked in the integrity label of the connection. Second, we check that the integrity label $I(\ell_n)$ of the original network connection is a lower bound for both the integrity label $I(\ell_{ci})$ of the new cookies set in the browser and the information $\Gamma_I(u)$, so that low-integrity connections cannot be used to set high-integrity cookies or to send requests to security-sensitive URLs.

Rule (T-Get) ensures that a cookie with confidentiality label $C(\ell_t)$ can only be accessed by a reader with confidentiality label $C(\ell_r) \subseteq C(\ell_t)$; dually, a cookie with integrity label $I(\ell_t)$ can only be accessed by a reader with integrity label $I(\ell_r) \supseteq I(\ell_t)$. The first condition guarantees that high-confidentiality cookies are only disclosed to their intended readers, while the second condition ensures that low-integrity cookies cannot affect the execution of high-integrity scripts.

Rule (T-Set) is dual to rule (T-Get). It checks that a cookie with confidentiality label $C(\ell_t)$ can only be set by a writer with confidentiality label $C(\ell_w) \supseteq C(\ell_t)$; dually, a cookie with integrity label $I(\ell_t)$ can only be set by a reader with integrity label $I(\ell_w) \subseteq I(\ell_t)$. The first condition guarantees that high-confidentiality scripts cannot reveal their secrets by setting low-confidentiality cookies, while the second condition ensures that high-integrity cookies can only be set or modified by their intended writers.

Rule (T-Send) enforces similar invariants to the rules for redirections (T-DocRedir) and (T-XhrRedir). The only difference with respect to these rules is that, in contrast to HTTP(S) redirects, the sending of an AJAX request does not directly set new cookies in the browser, so the conditions to check are slightly more concise.

Having defined a constraint system for transfer functions, we now formalize the non-interference properties that are supported by it. The following theorems are similar to the results presented in Section 4.5.3, but instead of talking about the canonical transfer function we now talk of all the transfer functions that satisfy the constraints.

Theorem 5 (Confidentiality)**.** *Assume that* $\Gamma \vdash$ *transfer and let* $\pi_C = \langle rel_\ell, \sim_\ell \rangle$ *be the confidentiality policy such that:*

1. $\forall i : \neg rel_\ell(i) \triangleq i = \mathsf{load}(u) \wedge \Gamma_C(u) \cap C(\ell) = \emptyset$;
2. $\forall i, i' : i \sim_\ell i' \Leftrightarrow erase^C_\ell(i) = erase^C_\ell(i')$.

Then, FF^τ *is non-interferent under* π_C *when implementing transfer.*

Theorem 6 (Integrity)**.** *Assume that* $\Gamma \vdash$ *transfer and let* $\pi_I = \langle rel_\ell, \sim_\ell \rangle$ *be the integrity policy such that:*

1. $\forall o : rel_\ell(o) \triangleq o = \mathsf{net\_req}(u : CK) \wedge \Gamma_I(u) \cap I(\ell) = \emptyset$;
2. $\forall o, o' : o \sim_\ell o' \Leftrightarrow erase^I_\ell(o) = erase^I_\ell(o')$.

Then, FF^τ *is non-interferent under* π_I *when implementing transfer.*

As explained in the proof sketch in Section 4.5.3, we show that the canonical transfer function always satisfies the constraints, in order to obtain the non-interference results for the canonical transfer function.

Theorem 7. *If* $\Gamma, f \rhd transfer(\mathsf{event\_type}, \tau_1, \tau_2) \rightsquigarrow \vec{\ell}$*, then* $\Gamma \vdash transfer(\mathsf{event\_type}, \tau_1, \tau_2) = \vec{\ell}$.

Proof. By a case analysis on the rule applied to prove the judgement in the premise and an application of the corresponding constraint-check, using the well-formedness of f. □

A.1.3 Failure Semantics

Table A.4 shows an excerpt of the failure semantics. It is used in the compatibility theorem to make failing runtime checks in the transfer function explicit in the output stream.

As opposed to the original semantics of FF^τ, the failure semantics contains multiple rules per event - one rule for the case in which no failure occurs and one rule for each possible failure of the transfer function. We elaborate this on the example of a load-event. For a load-event there are two possible sources of failure:

1. The transfer function can fail for the load-event, because the runtime check in rule (G-Load) fails.
2. The transfer function can fail for the get-event, because one of the runtime checks in rule (G-Get) fails for one cookie while retrieving the cookies from the cookie jar.

The rule (I-Load1) treats the case in which none of the failures occurs (this is ensured by the premise). The rule is then equivalent to the original rule (I-Load). The rule (I-Load2) covers the case of the failure described in 1) while the rule (I-Load3) covers the case of the failure described in 2). The design of the rules for other events is analogous.

We now give the proof to Theorem 4.

Theorem 4 (Compatibility)**.** *Let C_0 be the initial state of $\mathsf{FF}^{\tau}_{\star}(\Gamma_\top, id)$ and assume that the function $\kappa : \mathcal{D} \times \mathcal{S} \to$ Tags assigns the top label $\top$ to all the elements of its domain. If $C_0(I) \Downarrow O$, then $\star$ does not occur in O.*

Proof. By a case analysis of the reduction rules, one can easily verify that $C(\ell_n) = C(\ell_s) = \top_s$ for all labels of network connections and scripts, so the confidentiality checks in rule (G-Load), (G-DocRedir), (G-XhrRedir) and (G-Send) trivially succeed. We can also observe $C(\ell_{ci}) = \top_s$ for all cookie writes, so the confidentiality condition on (G-Set) is always true. Because of κ, we know $C(\tau) = \top_s$ for all cookies $\mathsf{ck}(k, v)^\tau$ in the cookie jar, and hence the confidentiality condition on (G-Get) holds true. As we know $\Gamma_I(u) = \top_s$ for all u, the integrity checks in (G-DocRedir), (G-XhrRedi) and (G-Send) succeed. One can observe that we have $I(\ell_{co}) = \top_s$ for cookie reads, hence the integrity check in (G-Get) succeeds as well. Because of κ, we know $I(\tau) = \top_s$ for all cookies $\mathsf{ck}(k, v)^\tau$ and hence the integrity condition in (G-Set) holds true. □

A.1.4 Additional Examples

Cookie Protection Against Network Attackers

The Secure attribute is the standard defense mechanism for authentication cookies against network attackers [Bar11]. If a cookie is marked as Secure, the browser will only attach it to HTTPS requests, thus ensuring that the cookie is never sent in clear. The Secure attribute does not provide strong integrity guarantees, since Secure cookies set over HTTPS can be overwritten by non-Secure cookies set over HTTP [ZJL+15].

It is indeed much harder to protect cookies against powerful attackers, like network attackers. Based on the previous informal explanation, it seems natural to model the Secure attribute in our framework by giving cookies set by the domain d the following label: $\ell_c = (\{\mathsf{https}(d)\}, \{\mathsf{http}(d), \mathsf{https}(d)\})$. However, if we want to let scripts access these cookies, as it normally happens for Secure cookies, the price to pay for non-interference is high: in particular, scripts downloaded from the domain d cannot perform *any* network communication at all. Technically, this is a consequence of the observation that scripts accessing these cookies must have a label ℓ such that $C(\ell) \subseteq \{\mathsf{https}(d)\}$ by rule (G-Get), but any attempt to communicate with a URL u by these scripts would fail by rule (G-Send), given that the rule requires $\mathit{evt\_label}(u) \subseteq C(\ell)$, but $\mathit{evt\_label}(u) \not\subseteq \{\mathsf{https}(d)\}$ for all u. Though this looks restrictive, it is actually correct, since the *presence* of any network communication is visible to a network attacker, hence it may act as a side-channel to leak the cookie value. As an extreme but simple example, a script may branch over a conditional, checking whether the value of the cookie is equal to a given string, and only send a bit over the network if this is true, thus revealing the cookie value.

If scripts do not need to access authentication cookies, which is the most common case for web applications, protection against network attackers can be enforced by raising the label of scripts to $\top$. This solution would not constrain network communication and would mimic the behaviour of standard cookies marked as both HttpOnly and Secure, though with the notable caveats on integrity discussed for HttpOnly cookies. Better protection against active network attackers can be implemented by changing the integrity label of cookies to just $\{\mathsf{https}(d)\}$, thus preventing cookies from being set or overwritten over HTTP.

Preventing Cross-Site Request Forgery

It is easy to provide protection against CSRF attacks in our framework, since the format of our labelling immediately supports specifications in the spirit of Allowed Referrer Lists (ARLs) [CMKW13], but with better security guarantees. ARLs allow individual websites to specify which URLs are legitimately entitled to send authenticated requests to them: if the browser attempts to contact a website from an address which is not included in the ARL specified by the website, the authentication cookies are not attached.

Moving to our framework, assume that a URL u should only accept authenticated requests from its own domain d and another domain d'. We can specify a labelling Γ such that $\Gamma_I(u) = \{\mathsf{http}(d), \mathsf{https}(d), \mathsf{http}(d'), \mathsf{https}(d')\}$. Notice that the protection granted by this labelling is quite strong: not only it ensures that authenticated requests to u can only be sent by pages hosted on d or d', but it also guarantees that the attacker cannot force d or d' into abusing their privileges by exploiting reflected XSS attacks enabled by HTTP(S) redirects.

To exemplify, let u_a be a HTTP URL pointing to the attacker website and u_d be a HTTP URL on the domain d, and pick the following input stream:

$$I \quad = \quad [\mathsf{load}(u_a), \mathsf{doc\_redir}_n(u_a : \emptyset, u_d), \mathsf{doc\_resp}_n(u_d : \emptyset, \mathsf{xhr}(u, \lambda x.\mathsf{unit}))].$$

The input stream I models the behaviour of an attacker abusing an HTTP(S) redirection from u_a to u_d to mount a reflected XSS attack on u_d, which then attempts to force the page hosted at u_d into sending an authenticated request to u. This request, however, will not be sent under the labelling above, though it is fired from the domain d. The reason is that the network connection n instantiated when processing the $\mathsf{load}(u_a)$ event is initially given an integrity label $\{\mathsf{http}(d_a)\}$, where d_a is the attacker-controlled domain hosting the malicious page at u_a. The redirection then changes the integrity label of n to $\{\mathsf{http}(d_a), \mathsf{http}(d)\}$ and this label is inherited by the script running in the origin u_d. Since $\{\mathsf{http}(d_a), \mathsf{http}(d)\} \not\subseteq \Gamma_I(u)$, the XHR request is dropped. Protecting from this class of attacks is beyond the capabilities of ARLs, since the last malicious request would be sent by an allowed referrer.

A.2 Proofs

A.2.1 Preliminaries

The proof of the following results is given in [BPS+09].

Definition 12 (Unwinding Relation)**.** *An* unwinding relation *is a label-indexed family of binary relations $\mathcal{R}_\ell$ on states of a reactive system with the following properties:*

1. *if $Q\,\mathcal{R}_\ell\,Q'$, then $Q'\,\mathcal{R}_\ell\,Q$;*
2. *if $C\,\mathcal{R}_\ell\,C'$ and $C \xrightarrow{i} P$ and $C' \xrightarrow{i'} P'$ and $i \sim_\ell i'$ with $rel_\ell(i)$ and $rel_\ell(i')$, then $P\,\mathcal{R}_\ell\,P'$;*
3. *if $C\,\mathcal{R}_\ell\,C'$ and $C \xrightarrow{i} P$ with $\neg rel_\ell(i)$, then $P\,\mathcal{R}_\ell\,C'$;*
4. *if $P\,\mathcal{R}_\ell\,C$ and $P \xrightarrow{o} Q$, then $\neg rel_\ell(o)$ and $Q\,\mathcal{R}_\ell\,C$;*
5. *if $P\,\mathcal{R}_\ell\,P'$, then either of the following conditions hold true:*
   a) *$P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ with $o \sim_\ell o'$ and $Q\,\mathcal{R}_\ell\,Q'$;*
   b) *$P \xrightarrow{o} Q$ with $\neg rel_\ell(o)$ and $Q\,\mathcal{R}_\ell\,P'$;*
   c) *$P' \xrightarrow{o'} Q'$ with $\neg rel_\ell(o')$ and $P\,\mathcal{R}_\ell\,Q'$.*

Theorem 8. *Let C_0 be the initial state of a reactive system R. If $C_0\,\mathcal{R}_\ell\,C_0$ for some unwinding relation $\mathcal{R}$, then R satisfies non-interference.*

A.2.2 Proof of Confidentiality

Definition 13 (Low Equivalence)**.** *We define an* erasure *operator $erase^C_\ell(\cdot)$ on different browser data structures:*

- *$erase^C_\ell(K)$ is the map with domain $dom(k)$ such that for all $d \in dom(K)$ we have $(erase^C_\ell(K))(d) = ck\text{-}erase^C_\ell(K(d))$*
- *$erase^C_\ell(N)$ is the map obtained from N by erasing all the entries $n^\tau : u$ such that $C(\tau) \cap C(\ell) = \emptyset$;*
- *$erase^C_\ell(H)$ is the map obtained from H by erasing all the entries $n^\tau : (u, \lceil \lambda x.e \rfloor_{@u'})$' such that $C(\tau) \cap C(\ell) = \emptyset$;*
- *$erase^C_\ell(T) =$ wait whenever $T = \lceil e \rfloor^\tau_{@u}$ with $C(\tau) \cap C(\ell) = \emptyset$, while $erase^C_\ell(T) = T$ otherwise.*

We then define a binary low equivalence *relation $\simeq^C_\ell$ between data structures coinciding after applying the erasure $erase^C_\ell(\cdot)$.*

Definition 14 (Candidate Relation for Confidentiality)**.** *Let $Q = \langle K, N, H, T, O\rangle$ and let $Q' = \langle K', N', H', T', O'\rangle$. We write $Q\,\mathcal{R}^C_\ell\,Q'$ if and only if: (1) $K \simeq^C_\ell K'$; (2) $N \simeq^C_\ell N'$; (3) $H \simeq^C_\ell H'$; (4) $T \simeq^C_\ell T'$; (5) $O \approx_\ell O'$.*

Lemma 1. *For the initial state C_0 we have $C_0\,\mathcal{R}^C_\ell\,C_0$.*

Proof. This follows directly from the reflexivity of $\simeq_\ell^C$ and $\approx_\ell$. □

Lemma 2. *$\mathcal{R}_\ell^C$ satisfies the first condition of Definition 12.*

Proof. A straightforward syntactic check on the definition of $\mathcal{R}_\ell^C$. □

Lemma 3 (Visible Cookies)**.** *Let $\Gamma \vdash$ transfer and $K \simeq_\ell^C K'$. If $C(\tau) \cap C(\ell) \neq \emptyset$, then $!K(u)^\tau = !K'(u)^\tau$ for any u.*

Proof. We only show $!K(u)^\tau \subseteq !K'(u)^\tau$, the proof of the other direction is analogous.

Let $\mathsf{ck}(k,v)^{\tau'} \in !K(u)^\tau$. Then we know that *transfer*$(\mathsf{get}, \tau, \tau')\downarrow$. By the constraints of the transfer function we know from the rule (T-Get) that $C(\tau) \subseteq C(\tau')$. It follows that $C(\tau') \cap C(\ell) \neq \emptyset$ and hence $\mathsf{ck}(k,v)^{\tau'} \in !(\mathit{erase}_\ell^C(K'))(u)^\tau$ by the definition of $\mathit{erase}_\ell^C(K)$. Because of $K \simeq_\ell^C K'$, we also have $\mathsf{ck}(k,v)^{\tau'} \in !(\mathit{erase}_\ell^C(K'))(u)^\tau$ and hence $\mathsf{ck}(k,v)^{\tau'} \in !K'(u)^\tau$. □

Lemma 4 (Visible Updates)**.** *Let $K \simeq_\ell^C K'$. If ck-erase$_\ell^C(CK)$ = ck-erase$_\ell^C(CK')$, then $K \xleftarrow{CK}_\tau u \simeq_\ell^C K' \xleftarrow{CK'}_\tau u$ for any u, τ.*

Proof. In order to prove the claim we have to show that for all domains d the following holds:

$$\mathit{ck\text{-}erase}_\ell^C((K \xleftarrow{CK}_\tau u)(d)) = \mathit{ck\text{-}erase}_\ell^C((K' \xleftarrow{CK'}_\tau u)(d))$$

We distinguish two cases:

- If $d \neq \mathit{host}(u)$ then we have $K(d) = (K \xleftarrow{CK}_\tau u)(d)$ and $K'(d) = (K' \xleftarrow{CK'}_\tau u)(d)$. As we know $\mathit{ck\text{-}erase}_\ell^C(K(d)) = \mathit{ck\text{-}erase}_\ell^C(K'(d))$ by the definition of $K \simeq_\ell^C K'$ we can conclude $\mathit{ck\text{-}erase}_\ell^C((K \xleftarrow{CK}_\tau u)(d)) = \mathit{ck\text{-}erase}_\ell^C((K' \xleftarrow{CK'}_\tau u)(d))$.
- If $d = \mathit{host}(u)$ then let $\mathsf{ck}(k_1,v_1)^{\tau_1} \in \mathit{ck\text{-}erase}_\ell^C((K \xleftarrow{CK}_\tau u)(d))$. This then implies $\mathsf{ck}(k_1,v_1)^{\tau_1} \in (K \xleftarrow{CK}_\tau u)(d)$ and $C(\tau_1) \cap C(\ell) \neq \emptyset$.

  We do a case distinction following the definition of $(K \xleftarrow{CK}_\tau u)(d)$:

  - If $\mathsf{ck}(k_1,v_1)^{\tau_1} \in CK$ with *transfer*$(\mathsf{set}, \tau, \tau_1)\downarrow$, then because of $\mathit{ck\text{-}erase}_\ell^C(CK) = \mathit{ck\text{-}erase}_\ell^C(CK')$ and $C(\tau_1) \cap C(\ell) \neq \emptyset$ we know that $\mathsf{ck}(k_1,v_1)^{\tau_1} \in CK'$, hence $\mathsf{ck}(k_1,v_1)^{\tau_1} \in (K' \xleftarrow{CK'}_\tau u)(d)$.
  - If $\mathsf{ck}(k_1,v_1)^{\tau_1} \in K(d)$ and for all $\mathsf{ck}(k_2,v_2)^{\tau_2} \in CK$ we have $k_1 \neq k_2$ or we have *transfer*$(\mathsf{set}, \tau, \tau_2)\uparrow$ then let $\mathsf{ck}(k_3,v_3)^{\tau_3} \in CK'$. We have to show that $k_3 \neq k_1$ or *transfer*$(\mathsf{set}, \tau, \tau_3)\uparrow$. We perform a case distinction:
    - If $k_3 \neq k_1$ then the claim follows trivially.
    - If $k_3 = k_1$ then we observe that $\tau_3 = \kappa(d,k_3) = \kappa(d,k_1) = \tau_1$. This implies $C(\tau_3) \cap C(\ell) \neq \emptyset$ and hence $\mathsf{ck}(k_3,v_3)^{\tau_3} \in CK$ because $\mathit{ck\text{-}erase}_\ell^C(CK) = \mathit{ck\text{-}erase}_\ell^C(CK')$. We then know *transfer*$(\mathsf{set}, \tau, \tau_3)\uparrow$, which concludes the proof.

□

Lemma 5 (Invisible Updates)**.** *Let* $\Gamma \vdash$ *transfer and* $C(\tau) \cap C(\ell) = \emptyset$*. Then* $K \simeq_\ell^C K \xleftarrow{CK}_\tau u$ *for all* K, u, CK*.*

Proof. We show both directions of set inclusions:

- Let $\mathsf{ck}(k_1, v_1)^{\tau_1} \in \mathit{erase}_\ell^C(K(d))$. This implies $C(\tau_1) \cap C(\ell) \neq \emptyset$. Let $\mathsf{ck}(k_2, v_2)^{\tau_2} \in CK$. To show $\mathsf{ck}(k_1, v_1)^{\tau_1} \in \mathit{erase}_\ell^C((K \xleftarrow{CK}_\tau u)(d))$ we have to show $k_2 \neq k_1$ or $\mathit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$. We perform a case distinction:
  - if $k_2 \neq k_1$ then the claim follows trivially.
  - If $k_2 = k_1$ then we observe that $\tau_2 = \kappa(d, k_2) = \kappa(d, k_1) = \tau_1$. This implies $C(\tau_2) \cap C(\ell) \neq \emptyset$, hence $C(\tau_2) \not\subseteq C(\tau)$ and by the constraints we have $\mathit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$.
- Let $\mathsf{ck}(k_1, v_1)^{\tau_1} \in \mathit{erase}_\ell^C((K \xleftarrow{CK}_\tau u)(d))$. This implies $C(\tau_1) \cap C(\ell) \neq \emptyset$. We show that for all $\mathsf{ck}(k_2, v_2)^{\tau_2} \in CK$ we have $k_2 \neq k_1$ or $\mathit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$. We do a case distinction:
  - If $k_2 \neq k_1$ then the claim follows trivially.
  - If $k_2 = k_1$ then we observe that $\tau_2 = \kappa(d, k_2) = \kappa(d, k_1) = \tau_1$. This implies $C(\tau_2) \cap C(\ell) \neq \emptyset$, hence $C(\tau_2) \not\subseteq C(\tau)$ and we have $\mathit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$.

□

Lemma 6. $\mathcal{R}_\ell^C$ *satisfies the second condition of Definition 12.*

Proof. Let $C = \langle K, N, H, \mathsf{wait}, [\,]\rangle$ and $C' = \langle K', N', H', \mathsf{wait}, [\,]\rangle$ with $C\,\mathcal{R}_\ell^C\,C'$. By definition of $\mathcal{R}_\ell^C$, we have:

(1) $K \simeq_\ell^C K'$;

(2) $N \simeq_\ell^C N'$;

(3) $H \simeq_\ell^C H'$.

Assume that $C \xrightarrow{i} P$ and $C' \xrightarrow{i'} P'$ with $i \sim_\ell i'$ and $\mathit{rel}_\ell(i)$ and $\mathit{rel}_\ell(i')$. We perform a case distinction on i:

- if $i = \mathsf{load}(u)$, then $i' = \mathsf{load}(u)$ by definition of $\sim_\ell$. Given that we assume $\mathit{rel}_\ell(i)$, we have $\Gamma_C(u) \cap C(\ell) \neq \emptyset$. By the reduction rules, assuming that $\mathit{transfer}(\mathsf{load}, \mathit{tag}(u), -) = (\ell_n, -, \ell_{co}, -)$, we then have:

$$\begin{aligned} P &= \langle K, N \uplus \{n^{\ell_n} : u\}, H, \mathsf{wait}, \mathsf{doc\_req}(u : !K(u)^{\ell_{co}})\rangle \\ P' &= \langle K', N' \uplus \{n^{\ell_n} : u\}, H', \mathsf{wait}, \mathsf{doc\_req}(u : !K'(u)^{\ell_{co}})\rangle \end{aligned}$$

  To prove the desired conclusion, we need to show:

(a) $N \uplus \{n^{\ell_n} : u\} \simeq_\ell^C N' \uplus \{n^{\ell_n} : u\}$

(b) $\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}) \approx_\ell \mathsf{doc\_req}(u : !K'(u)^{\ell_{co}})$

Point (a) is an immediate consequence of (2), while point (b) is more complicated. We distinguish three sub-cases:

- if $\mathit{evt\_label}(u) \not\subseteq C(\ell)$, then $\neg \mathit{rel}_\ell(\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}))$ and $\neg \mathit{rel}_\ell(\mathsf{doc\_req}(u : !K'(u)^{\ell_{co}}))$, hence we conclude by applying rules (S-Left), (S-Right) and (S-Empty);
- if $\mathit{evt\_label}(u) \subseteq C(\ell)$ and $\mathit{msg\_label}(u) \not\subseteq C(\ell)$, then $\mathit{rel}_\ell(\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}))$ and $\mathit{rel}_\ell(\mathsf{doc\_req}(u : !K'(u)^{\ell_{co}}))$.
  However, we also know $\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}) \sim_\ell \mathsf{doc\_req}(u : !K'(u)^{\ell_{co}})$, hence we conclude by applying rule (S-Match);
- if $\mathit{evt\_label}(u) \subseteq C(\ell)$ and $\mathit{msg\_label}(u) \subseteq C(\ell)$, then $\mathit{rel}_\ell(\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}))$ and $\mathit{rel}_\ell(\mathsf{doc\_req}(u : !K'(u)^{\ell_{co}}))$. By the constraints of the transfer function, we know that $\mathit{msg\_label}(u) \subseteq C(\ell_{co})$. This implies $C(\ell_{co}) \cap C(\ell) \neq \emptyset$ and hence $!K(u)^{\ell_{co}} = !K'(u)^{\ell_{co}}$ by using (1) and Lemma 3, which is enough to conclude by rule (S-Match), using the reflexivity of the $\sim_\ell$ relation;

- if $i = \mathsf{doc\_resp}_n(u : CK, e)$, then $i' = \mathsf{doc\_resp}_n(u : CK', e)$ with $\mathit{ck\text{-}erase}_\ell^C(CK) = \mathit{ck\text{-}erase}_\ell^C(CK')$ by definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:
  - assume that rule (I-DocResp) is used on both C and C', then we know that $N = N_0 \uplus \{n^{\tau_n} : u\}$ and $N' = N_1 \uplus \{n^{\tau'_n} : u\}$ for some N_0, N_1 such that $N_0 \simeq_\ell^C N_1$. Assuming that we have
    - $\mathit{transfer}(\mathsf{doc\_resp}, \tau_n, \mathit{tag}(u)) = (-, \tau_{ci}, -, \tau_s)$ and
    - $\mathit{transfer}(\mathsf{doc\_resp}, \tau'_n, \mathit{tag}(u)) = (-, \tau'_{ci}, -, \tau'_s)$

    , we then get:
    $$\begin{aligned} P &= \langle K \xleftarrow{CK}_{\tau_{ci}} u, N_0, H, \lceil e \rfloor_{@u}^{\tau_s}, [\,] \rangle \\ P' &= \langle K' \xleftarrow{CK'}_{\tau'_{ci}} u, N_1, H', \lceil e \rfloor_{@u}^{\tau'_s}, [\,] \rangle \end{aligned}$$
    To prove the desired conclusion, we need to show:
    (a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq_\ell^C K' \xleftarrow{CK'}_{\tau'_{ci}} u$.
    (b) $\lceil e \rfloor_{@u}^{\tau_s} \simeq_\ell^C \lceil e \rfloor_{@u}^{\tau'_s}$
    We distinguish two subcases:
    - if $C(\tau_n) \cap C(\ell) \neq \emptyset$ then we know by (2) and the definition of $\simeq_\ell^C$ that $\tau_n = \tau'_n$. Hence we also have $\tau_{ci} = \tau'_{ci}$ and $\tau_s = \tau'_s$. Point (a) follows directly from Lemma 4 and point (b) follows from the reflexivity of $\simeq_\ell^C$.
    - if $C(\tau_n) \cap C(\ell) = \emptyset$ then by (2) we also know that $C(\tau'_n) \cap C(\ell) = \emptyset$. By the constraints we know that $C(\tau) \cap C(\ell) = \emptyset$ for all $\tau \in \{\tau_s, \tau'_s, \tau_{ci}, \tau'_{ci}\}$. Point (a) then follows from Lemma 5, (1) and transitivity and point (b) follows from the definition of $\simeq_\ell^C$.

– assume that rule (I-DocResp) is used on C, while C' fails using rule (I-Complete). This implies that $N = N_0 \uplus \{n^{\tau_n} : u\}$ for some N_0 with $N_0 \simeq_\ell^C N'$ and $C(\tau_n) \cap C(\ell) = \emptyset$. Assuming that $\mathit{transfer}(\mathsf{doc\_resp}, \tau_n, \mathit{tag}(u)) = (-, \tau_{ci}, -, \tau_s)$, we have:

$$\begin{array}{lcl} P & = & \langle K \xleftarrow{CK}_{\tau_{ci}} u, N_0, H, \lceil e \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

To prove the desired conclusion we need to show:

(a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq_\ell^C K'$
(b) $\lceil e \rfloor^{\tau_s}_{@u} \simeq_\ell^C \mathsf{wait}$
(c) $[\,] \approx_\ell \bullet$

By the constraints we know that $C(\tau) \cap C(\ell) = \emptyset$ for all $\tau \in \{\tau_s, \tau_{ci}, \}$. Point (a) then follows from Lemma 5, (1) and transitivity and point (b) follows from the definition of $\simeq_\ell^C$. Point (c) follows from the invisibility of $\bullet$ and using (S-Right) and (S-Empty).

– assume that rule (I-DocResp) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

– assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{lcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

- if $i = \mathsf{doc\_redir}_n(u : CK, u')$, then $i' = \mathsf{doc\_redir}_n(u : CK', u')$ with $\mathit{ck\text{-}erase}_\ell^C(CK) = \mathit{ck\text{-}erase}_\ell^C(CK')$ by definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:

  – assume that rule (I-DocRedir) is used on both C and C', then we know that $N = N_0 \uplus \{n^{\tau_n} : u\}$ and $N' = N_1 \uplus \{n^{\tau'_n} : u\}$ for some N_0, N_1 such that $N_0 \simeq_\ell^C N_1$. Assuming

    * $\mathit{transfer}(\mathsf{doc\_redir}, \tau_n, \mathit{tag}(u'), -) = (\tau_m, \tau_{ci}, \tau_{co}, -)$,
    * $\mathit{transfer}(\mathsf{doc\_redir}, \tau'_n, \mathit{tag}(u')) = (\tau'_m, \tau'_{ci}, \tau'_{co}, -)$,
    * $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$
    * $K_1 = K' \xleftarrow{CK'}_{\tau'_{ci}} u$

    we then get

$$\begin{array}{lcl} P & = & \langle K_0, N_0 \uplus \{n^{\tau_m} : u'\}, H, \mathsf{wait}, \mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \rangle \\ P' & = & \langle K_1, N_1 \uplus \{n^{\tau'_m} : u'\}, H', \mathsf{wait}, \mathsf{doc\_req}(u' : !K_1(u')^{\tau'_{co}}) \rangle \end{array}$$

    To prove the desired conclusion, we need to show:

(a) $K_0 \simeq_\ell^C K_1$
(b) $N_0 \uplus \{n^{\tau_m} : u'\} \simeq_\ell^C N_1 \uplus \{n^{\tau'_m} : u'\}$
(c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell \mathsf{doc\_req}(u' : !K_1(u')^{\tau'_{co}})$

We distinguish two subcases:

* if $C(\tau_n) \cap C(\ell) \neq \emptyset$ then we know by (2) and the definition of $\simeq_\ell^C$ that $\tau_n = \tau'_n$. Hence we also have $\tau_m = \tau'_m$, $\tau_{ci} = \tau'_{ci}$ and $\tau_{co} = \tau'_{co}$. Point (a) follows directly from Lemma 4 and point (b) follows from (2) and the definition of $\simeq_\ell^C$. To show point (c) we do an additional case distinction:
  · if $\mathit{evt\_label}(u') \not\subseteq C(\ell)$ then we have $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_1(u')^{\tau_{co}}))$, hence we conclude by applying rules (S-Left) and (S-Right).
  · If $\mathit{evt\_label}(u') \subseteq C(\ell)$ and $\mathit{msg\_label}(u') \not\subseteq C(\ell)$ then the claim follows from the definition of $\sim_\ell^C$ and the rule (S-Match).
  · If $\mathit{evt\_label}(u') \subseteq C(\ell)$ and $\mathit{msg\_label}(u') \subseteq C(\ell)$ then we also get $\mathit{msg\_label}(u') \subseteq C(\tau_{co})$ by the constraints of the transfer-function and the rule (T-DocRedir). This implies $C(\tau_{co}) \cap C(\ell) \neq \emptyset$ and hence the we know $!K_0(u')^{\tau_{co}} = !K_1(u')^{\tau_{co}}$ by Lemma 3. Point (c) then follows from the definition of $\sim_\ell$ and rule (S-Match).
* if $C(\tau_n) \cap C(\ell) = \emptyset$ then by (2) we know that also $C(\tau'_n) \cap C(\ell) = \emptyset$.
  By the constraints of the transfer-function we get $C(\tau) \cap C(\ell)$ for all $\tau \in \{\tau_m, \tau'_m, \tau_{ci}, \tau'_{ci}\}$ by rule (T-DocRedir). Point (a) follows from Lemma 5, (1) and transitivity and point (b) follows from (2) and the definition of $\simeq_\ell^C$. The constraints also give us $\mathit{evt\_label}(u') \cap C(\ell) = \emptyset$ and hence we know $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_1(u')^{\tau'_{co}}))$. We can conclude (c) using rules (S-Left) and (S-Right) and (S-Empty).

– assume that rule (I-DocRedir) is used on C, while C' fails using rule (I-Complete). This implies that $N = N_0 \uplus \{n^{\tau_n} : u\}$ for some N_0 with $N_0 \simeq_\ell^C N'$ and $C(\tau_n) \cap C(\ell) = \emptyset$. Assuming that $\mathit{transfer}(\mathsf{doc\_redir}, \tau_n, \mathit{tag}(u')) = (\tau_m, \tau_{ci}, \tau_{co}, -)$ and $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$ we have:

$$\begin{aligned} P &= \langle K_0, N_0 \uplus \{n^{\tau_m} : u'\}, H, \mathsf{wait}, \mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}})\rangle \\ P' &= \langle K', N', H', \mathsf{wait}, \bullet\rangle \end{aligned}$$

To prove the desired conclusion we need to show:

(a) $K_0 \simeq_\ell^C K'$
(b) $N_0 \uplus \{n^{\tau_m} : u'\} \simeq_\ell^C N'$
(c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell \bullet$

By the constraints of the transfer-function we get $C(\tau) \cap C(\ell)$ for all $\tau \in \{\tau_m, \tau_{ci}\}$ by rule (T-DocRedir). Point (a) follows from Lemma 5, (1) and transitivity and point (b) follows from (2) and the definition of $\simeq_\ell^C$. The constraints also give us $\mathit{evt\_label}(u') \cap C(\ell) = \emptyset$ and hence we know $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\bullet)$. We can conclude (c) using rules (S-Left) and (S-Right) and (S-Empty).

– assume that rule (I-DocRedir) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

– assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{rcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

- if $i = \mathsf{xhr\_resp}_n(u : CK, v)$, then $i' = \mathsf{xhr\_resp}_n(u : CK', v)$ with $\mathit{ck\text{-}erase}_\ell^C(CK) = \mathit{ck\text{-}erase}_\ell^C(CK')$ by definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:

  – assume that rule (I-XhrResp) is used on both C and C', then we know that $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u'})\}$ and $H' = H_1 \uplus \{n^{\tau'_n} : (u, \lceil \lambda x.e' \rfloor_{@u''})\}$ for some H_0, H_1 such that $H_0 \simeq_\ell^C H_1$. Assuming that $\mathit{transfer}(\mathsf{xhr\_resp}, \tau_n, \mathit{tag}(u)) = (-, \tau_{ci}, -, \tau_s)$ and $\mathit{transfer}(\mathsf{xhr\_resp}, \tau'_n, \mathit{tag}(u)) = (-, \tau'_{ci}, -, \tau'_s)$, we have:

$$\begin{array}{rcl} P & = & \langle K \xleftarrow[\tau_{ci}]{CK} u, N, H_0, \lceil e\{v/x\} \rfloor_{@u'}^{\tau_s}, [\,] \rangle \\ P' & = & \langle K' \xleftarrow[\tau'_{ci}]{CK'} u, N', H_1, \lceil e\{v/x\} \rfloor_{@u''}^{\tau'_s}, [\,] \rangle \end{array}$$

To prove the desired conclusion, we need to show:

(a) $K \xleftarrow[\tau_{ci}]{CK} u \simeq_\ell^C K' \xleftarrow[\tau'_{ci}]{CK'} u$.

(b) $\lceil e\{v/x\} \rfloor_{@u'}^{\tau_s} \simeq_\ell^C \lceil e'\{v/x\} \rfloor_{@u''}^{\tau'_s}$

We distinguish two subcases:

  * if $C(\tau_n) \cap C(\ell) \neq \emptyset$ then we know by (3) and the definition of $\simeq_\ell^C$ that $\tau_n = \tau'_n$, $u' = u''$ and $e = e'$. Hence we also have $\tau_{ci} = \tau'_{ci}$ and $\tau_s = \tau'_s$. Point (a) follows directly from Lemma 4 and point (b) follows from the definition of $\simeq_\ell^C$.
  * if $C(\tau_n) \cap C(\ell) = \emptyset$ then by (3) we know that also $C(\tau_n) \cap C(\ell) = \emptyset$. By the constraints we know that $C(\tau) \cap C(\ell) = \emptyset$ for all $\tau \in \{\tau_s, \tau'_s, \tau_{ci}, \tau'_{ci}\}$. Point (a) then follows from Lemma 5, (1) and transitivity and point (b) follows from the definition of $\simeq_\ell^C$.

  – assume that rule (I-XhrResp) is used on C, while C' fails using rule (I-Complete). This implies that $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u'})\}$ for some H_0 with $H_0 \simeq_\ell^C H'$ and $C(\tau_n) \cap C(\ell) = \emptyset$. Assuming that $\mathit{transfer}(\mathsf{xhr\_resp}, \tau_n, -) = (-, \tau_{ci}, -, \tau_s)$, we have:

$$\begin{array}{rcl} P & = & \langle K \xleftarrow[\tau_{ci}]{CK} u, N, H_0, \lceil e\{v/x\} \rfloor_{@u'}^{\tau_s}, [\,] \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

To prove the desired conclusion we need to show:

(a) $K \xleftarrow[\tau_{ci}]{CK} u \simeq_\ell^C K'$

(b) $\lceil e\{v/x\}\rfloor^{\tau_s}_{@u'} \simeq^C_\ell \mathsf{wait}$

(c) $[\,] \approx_\ell \bullet$

By the constraints we know that $C(\tau) \cap C(\ell) = \emptyset$ for all $\tau \in \{\tau_s, \tau_{ci}\}$. Point (a) then follows from Lemma 5, (1) and transitivity and point (b) follows from the definition of $\simeq^C_\ell$. Point (c) follows from $\neg \mathit{rel}(\bullet)$ and using rules (S-Right) and (S-Empty).

– assume that rule (I-XhrResp) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

– assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{lcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet\rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet\rangle \end{array}$$

The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

- if $i = \mathsf{xhr\_redir}_n(u : CK, u')$, then $i' = \mathsf{xhr\_redir}_n(u : CK', u')$ with $\mathit{ck\text{-}erase}^C_\ell(CK) = \mathit{ck\text{-}erase}^C_\ell(CK')$ by definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:

  – assume that rule (I-XhrRedir) is used on both C and C'. Then we know that $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u''})\}$ and $H' = H_1 \uplus \{n^{\tau'_n} : (u, \lceil \lambda x.e' \rfloor_{@u'''})\}$ for some H_0, H_1 such that $H_0 \simeq^C_\ell H_1$. Assuming that we have

    * $\mathit{transfer}(\mathsf{xhr\_redir}, \tau_n, \mathit{tag}(u'), -) = (\tau_m, \tau_{ci}, \tau_{co}, -)$,
    * $\mathit{transfer}(\mathsf{xhr\_redir}, \tau'_n, \mathit{tag}(u')) = (\tau'_m, \tau'_{ci}, \tau'_{co}, -)$,
    * $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$
    * $K_1 = K' \xleftarrow{CK'}_{\tau'_{ci}} u$

    we then get

$$\begin{array}{ll} P = & \langle K_0, N, H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}})\rangle \\ P' = & \langle K_1, N', H_1 \uplus \{n^{\tau'_m} : (u', \lceil \lambda x.e' \rfloor_{@u'''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K_1(u')^{\tau'_{co}})\rangle \end{array}$$

    To prove the desired conclusion, we need to show:

    (a) $K_0 \simeq^C_\ell K_1$

    (b) $H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\} \simeq^C_\ell H_1 \uplus \{n^{\tau'_m} : (u', \lceil \lambda x.e' \rfloor_{@u'''})\}$

    (c) $\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}) \sim_\ell \mathsf{xhr\_req}(u' : !K_1(u')^{\tau'_{co}})$

    We distinguish two subcases:

    * if $C(\tau_n) \cap C(\ell) \neq \emptyset$ then we know by (3) and the definition of erase^C_ℓ that $\tau_n = \tau'_n$, $e = e'$ and $u'' = u'''$. Hence we also have $\tau_m = \tau'_m$, $\tau_{ci} = \tau'_{ci}$ and $\tau_{co} = \tau'_{co}$. Point (a) follows directly from Lemma 4 and point (b) follows from (3) and the definition of $\simeq^C_\ell$. To show point (c) we do an additional case distinction:

· if $\mathit{evt\_label}(u') \not\subseteq C(\ell)$ then we have $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_1(u')^{\tau_{co}}))$, hence we conclude by applying rules (S-Left) and (S-Right).

· If $\mathit{evt\_label}(u') \subseteq C(\ell)$ and $\mathit{msg\_label}(u') \not\subseteq C(\ell)$ then the claim follows from the definition of $\sim_\ell^C$ and the rule (S-Match).

· If $\mathit{evt\_label}(u') \subseteq C(\ell)$ and $\mathit{msg\_label}(u') \subseteq C(\ell)$ then we get that $\mathit{msg\_label}(u') \subseteq C(\tau_{co})$ by the constraints of the transfer-function and the rule (T-XhrRedir). This implies $C(\tau_{co}) \cap C(\ell) \neq \emptyset$ and hence the we know $!K_0(u')^{\tau_{co}} = !K_1(u')^{\tau_{co}}$ by Lemma 3. Point (c) then follows from the definition of $\sim_\ell$ and rule (S-Match).

∗ if $C(\tau_n) \cap C(\ell) = \emptyset$ then we know by (3) and the definition of $\simeq_\ell^C$ that $C(\tau_n) \cap C(\ell) = \emptyset$. By the constraints of the transfer-function we get $C(\tau) \cap C(\ell)$ for all $\tau \in \{\tau_m, \tau'_m, \tau_{ci}, \tau'_{ci}\}$ by rule (T-XhrRedir). Point (a) follows from Lemma 5, (1) and transitivity and point (b) follows from (3) and the definition of $\simeq_\ell^C$. The constraints also give us $\mathit{evt\_label}(u') \cap C(\ell) = \emptyset$ and hence we know $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_1(u')^{\tau'_{co}}))$. We can conclude (c) using rules (S-Left) and (S-Right) and (S-Empty).

– assume that rule (I-XhrRedir) is used on C, while C' fails using rule (I-Complete). This implies that $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u''})\}$ for some H_0 with $H_0 \simeq_\ell^C H'$ and $C(\tau_n) \cap C(\ell) = \emptyset$. Assuming that $\mathit{transfer}(\mathsf{xhr\_redir}, \tau_n, \mathit{tag}(u')) = (\tau_m, \tau_{ci}, \tau_{co}, -)$ and $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$ we have

$$\begin{array}{lcl} P & = & \langle K_0, N, H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}) \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

To prove the desired conclusion we need to show:

(a) $K_0 \simeq_\ell^C K'$

(b) $H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\} \simeq_\ell^C H'$

(c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell \bullet$

By the constraints of the transfer-function we get $C(\tau) \cap C(\ell)$ for all $\tau \in \{\tau_m, \tau_{ci}\}$ by rule (T-XhrRedir). Point (a) follows from Lemma 5, (1) and transitivity and point (b) follows from (3) and the definition of $\simeq_\ell^C$. The constraints also give us $\mathit{evt\_label}(u') \cap C(\ell) = \emptyset$ and hence we know $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\bullet)$. We can conclude (c) using rules (S-Left) and (S-Right) and (S-Empty).

– assume that rule (I-XhrRedir) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

– assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{lcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

□

Lemma 7. *$\mathcal{R}_\ell^C$ satisfies the third condition of Definition 12.*

Proof. Let $C = \langle K, N, H, \mathsf{wait}, [\,]\rangle$ and $C' = \langle K', N', H', \mathsf{wait}, [\,]\rangle$ with $C\,\mathcal{R}_\ell^C\,C'$. By definition of $\mathcal{R}_\ell^C$, we have:

(1) $K \simeq_\ell^C K'$;

(2) $N \simeq_\ell^C N'$;

(3) $H \simeq_\ell^C H'$.

Assume that $C \xrightarrow{i} P$ with $\neg \mathit{rel}_\ell(i)$. The only case when $\neg \mathit{rel}_\ell(i)$ holds true is when $i = \mathsf{load}(u)$ with $\Gamma_C(u) \cap C(\ell) = \emptyset$. By the reduction rules, assuming that $\mathit{transfer}(\mathsf{load}, \mathit{tag}(u), -) = (\tau_n, -, \tau_{co}, -)$, we then have:

$$P = \langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, \mathsf{doc\_req}(u : !K(u)^{\tau_{co}})\rangle$$

To prove the desired conclusion, we need to show:

(a) $N \uplus \{n^{\tau_n} : u\} \simeq_\ell^C N'$

(b) $\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}) \approx_\ell [\,]$

By the constraints we know $C(\tau_n) \cap C(\ell) = \emptyset$ and point (a) follows from (2) and the definition of $\simeq_\ell^C$. By the constraints we also get $\mathit{evt\_label}(u) \cap C(\ell) = \emptyset$ and hence $\neg \mathit{rel}(\mathsf{doc\_req}(u : !K(u)^{\tau_{co}}))$. We can conclude (b) using (S-Left) and (S-Empty). □

Lemma 8 (Invisible Scripts)**.** *Let $P = \langle K, N, H, \lceil e \rfloor_{@u}^{\tau_s}, [\,]\rangle$ and let $P\,\mathcal{R}_\ell^C\,Q$ for some state Q. If $C(\tau_s) \cap C(\ell) = \emptyset$ and $P \xrightarrow{o} Q'$ for some o, Q', then $\neg \mathit{rel}_\ell(o)$ and $Q'\,\mathcal{R}_\ell^C\,Q$.*

Proof. Let $Q = \langle K', N', H', T', O'\rangle$. By definition of $\mathcal{R}_\ell^C$ we have

(1) $K \simeq_\ell^C K'$

(2) $N \simeq_\ell^C N'$

(3) $H \simeq_\ell^C H'$

(4) $\lceil e \rfloor_{@u}^{\tau_s} \simeq_\ell^C T'$

(5) $[\,] \simeq_\ell^C O'$

We do an induction on the term structure of e:
Induction Hypothesis: The claim holds for all $P = \langle K, N, H, \lceil e' \rfloor^{\tau_s}_{@u}, [\,] \rangle$ where e' is a subterm of e.

In the following subcases we will say that the claim follows trivially if the claim follows directly from $(1) - (5)$ and the fact that $\lceil e' \rfloor^{\tau_s}_{@u} \simeq^C_\ell T'$ for all e' because $C(\tau_s) \cap C(\ell) = \emptyset$.

- If $e = \lambda x.e'\ v$ then (O-App) is used and $Q' = \langle K, N, H, \lceil e'\{v/x\} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. The claim follows trivially.
- If $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ then we distinguish two cases:
  - If there exist K^*, N^*, H^*, e^* and o^* such that we have $\langle K, N, H, \lceil e_1 \rfloor^{\tau_s}_{@u}, [\,] \rangle \overset{o^*}{\longmapsto} \langle K^*, N^*, H^*, \lceil e^* \rfloor^{\tau_s}_{@u}, [\,] \rangle$ then (O-LetCtx) is used and $Q' = \langle K^*, N^*, H^*, \lceil \mathsf{let}\ x = e^*\ \mathsf{in}\ e_2 \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = o^*$. As e_1 is a subterm of e we get $K^* \simeq^C_\ell K'$, $N^* \simeq^C_\ell N'$, $H^* \simeq^C_\ell H'$ and $\neg \mathit{rel}(o^*)$ by the induction hypothesis. The claim then follows directly.
  - Otherwise rule (O-Complete) is used and the claim follows trivially.
- If $e = \mathsf{let}\ x = v\ \mathsf{in}\ e'$ then (O-Let) is used and $Q' = \langle K, N, H, \lceil e'\{v/x\} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. The claim follows trivially.
- If $e = \mathsf{get\text{-}ck}(k)$ then we distinguish two cases:
  - If there exist τ, v with $\mathsf{ck}(k, v)^\tau \in\ !K(u)^{\tau_s}$ then rule (O-GetCookie) is used and $Q' = \langle K, N, H, \lceil v \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. The claim follows trivially.
  - Otherwise rule (O-Complete) is used and the claim follows trivially.
- If $e = \mathsf{set\text{-}ck}(k, v)$ then rule (O-SetCookie) is used. Let $CK = \{\mathsf{ck}(k, v)^\tau \mid \tau = \kappa(\mathit{host}(u), k)\}$, then $Q' = \langle K \overset{CK}{\longleftarrow}_{\tau_s} u, N, H, \lceil \mathit{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. Using (1) and Lemma 5 we get $K \overset{CK}{\longleftarrow}_{\tau_s} u \simeq^C_\ell K'$. The claim then follows trivially.
- If $e = \mathsf{xhr}(u', \lambda x.e')$ then we distinguish two cases:
  - If $\mathit{transfer}(\mathsf{send}, \tau_s, \mathit{tag}(u')) = (\tau_n, -, \tau_{co}, -)$ for some τ_n, τ_{co} then we know that rule (O-Xhr) is used and $Q' = \langle K, N, H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \mathsf{xhr\_req}(u' : !K(u')^{\tau_{co}})$.
    By the constraints we get $C(\tau_n) \cap C(\ell) = \emptyset$ and $\mathit{evt\_label}(u) \cap C(\ell) = \emptyset$. We get $H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\} \simeq^C_\ell H'$ directly from the definition of $\simeq^C_\ell$ and (3). Since we have $\neg \mathit{rel}(o)$, the claim then follows trivially. .
  - Otherwise rule (O-Complete) is used and the claim follows trivially.
- For all other forms of e (O-Complete) is used and the claim follows trivially.

□

Lemma 9. *$\mathcal{R}_\ell^C$ satisfies the fourth condition of Definition 12.*

Proof. Let $P = \langle K, N, H, T, O\rangle$ and $C = \langle K', N', H', \mathsf{wait}, [\,]\rangle$, where either $T \neq \mathsf{wait}$ or $O = o :: O'$ for some o, O' by definition of producer state. Assume that $P\,\mathcal{R}_\ell^C\,C$, by definition we have:

(1) $K \simeq_\ell^C K'$

(2) $N \simeq_\ell^C N'$

(3) $H \simeq_\ell^C H'$

(4) $T \simeq_\ell^C \mathsf{wait}$

(5) $O \approx_\ell [\,]$

We now distinguish two sub-cases:

- if $O = o :: O'$ for some o, O', the only available reduction rule for P is rule (O-Flush), hence:
$$P \xrightarrow{o} \langle K, N, H, T, O'\rangle.$$
Using (5) and the definition of stream similarity, we have $\neg \mathit{rel}_\ell(o)$ and $O' \approx_\ell [\,]$, which is enough to close the case;
- otherwise, assume without loss of generality that $O = [\,]$, then $T = \lceil e \rfloor_{@u}^{\tau_s}$ for some e, u, τ_s. We observe that point (4) implies that $C(\tau_s) \cap C(\ell) = \emptyset$, hence we conclude by Lemma 8;

□

Lemma 10. *$\mathcal{R}_\ell^C$ satisfies the fifth condition of Definition 12.*

Proof. Let $P = \langle K, N, H, T, O\rangle$ and $P' = \langle K', N', H', T', O'\rangle$ with $P\,\mathcal{R}_\ell^C\,P'$. By definition of $\mathcal{R}_\ell^C$, we have:

1. $K \simeq_\ell^C K'$;
2. $N \simeq_\ell^C N'$;
3. $H \simeq_\ell^C H'$;
4. $T \simeq_\ell^C T'$;
5. $O \approx_\ell O'$.

We distinguish four sub-cases:

- if $O = o :: O_0$ and $O' = o' :: O_1$, the only available reduction rule is (O-Flush) for both P and P'. We distinguish three sub-cases:
  - If $\neg \mathit{rel}(o)$ then $O \approx_\ell O'$ is shown by the rule (S-Left) and we know $O_0 \approx_\ell O'$. We have $P \xrightarrow{o} Q$ for $Q = \langle K, N, H, T, O_0 \rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case b).
  - If $\mathit{rel}(o)$ and $\neg \mathit{rel}(o')$ then $O \approx_\ell O'$ is shown by the rule (S-Right) and we know $O \approx_\ell O_1$. We have $P' \xrightarrow{o} Q'$ for $Q' = \langle K', N', H', T', O_1 \rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case c).
  - If $\mathit{rel}(o)$ and $\mathit{rel}(o')$ then $O \approx_\ell O'$ is shown by the rule (S-Match) and we know $O_0 \approx_\ell O_1$ and $o \sim_\ell o'$. By the rule (O-Flush) we have $P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ for $Q = \langle K, N, H, T, O_1 \rangle$ and $Q' = \langle K', N', H', T', O_0 \rangle$. Using $(1), (2), (3)$ and (4), we can match case a).
- if $O = o :: O''$ and $O' = [\,]$, we know that $O \approx_\ell O'$ can only be shown using rule (S-Left), hence we have $\neg \mathit{rel}(o)$ and $O'' \approx_\ell [\,]$. We have $P \xrightarrow{o} Q$ for $Q = \langle K, N, H, T, O'' \rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case b).
- if $O = [\,]$ and $O' = o' :: O''$, we know that $O \approx_\ell O'$ can only be shown using rule (S-Right), hence we have $\neg \mathit{rel}(o')$ and $[\,] \approx_\ell O''$. We have $P' \xrightarrow{o'} Q'$ for $Q' = \langle K', N', H', T', O'' \rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case c).
- if $O = O' = [\,]$, then by definition of producer state we have $T = \lceil e \rfloor^{\tau_s}_{@u}$ and $T' = \lceil e' \rfloor^{\tau'_s}_{@u'}$. We distinguish two sub-cases:
  - If $C(\tau_s) \cap C(\ell) = \emptyset$ and $P \xrightarrow{o} Q$ then we get $\neg \mathit{rel}(o)$ and $Q \, \mathcal{R}^C_\ell \, P'$ by Lemma 8 and we can match case b).
  - If $C(\tau_s) \cap C(\ell) \neq \emptyset$ then point (4) implies $e = e'$, $u = u'$ and $\tau_s = \tau'_s$.

    Assume $P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$. We show the following stronger claim, that is directly implying a): $o \sim_\ell o'$ and $Q \, \mathcal{R}^C_\ell \, Q'$ and $P \overset{o}{\mapsto} Q \iff P \overset{o'}{\mapsto} Q'$

    We do an induction on the term structure of e:
    Induction Hypothesis: The claim holds for all $P = \langle K, N, H, \lceil e'' \rfloor^{\tau_s}_{@u}, O \rangle$ and $P' = \langle K', N', H', \lceil e'' \rfloor^{\tau_s}_{@u}, O' \rangle$ where e'' is a subterm of e.
    - If $e = \lambda x.e_1 \; v$ then (O-App) is used on P and P' and

      $$\begin{aligned} Q &= \langle K, N, H, \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ Q' &= \langle K', N', H', \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,] \rangle \end{aligned}$$

      and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.
    - If $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ then we distinguish two cases:
      - If there exist K^*, N^*, H^*, e^* and o^* such that $\langle K, N, H, \lceil e_1 \rfloor^{\tau_s}_{@u}, [\,] \rangle \overset{o^*}{\mapsto} \langle K^*, N^*, H^*, \lceil e^* \rfloor^{\tau_s}_{@u}, [\,] \rangle$ then as e_1 is a subterm of e by the induction hypothesis we know that there exist $K^{**}, N^{**}, H^{**}, e^{**}$ and o^{**} such that

$\langle K', N', H', \lceil e_1 \rfloor^{\tau_s}_{@u}, [\,]\rangle \xrightarrow{o^{**}} \langle K^{**}, N^{**}, H^{**}, \lceil e^{**} \rfloor^{\tau_s}_{@u}, [\,]\rangle$ such that $o^* \sim_\ell o^{**}$, $K^* \simeq^C_\ell K^{**}$, $N^* \simeq^C_\ell N^{**}$, $H^* \simeq^C_\ell H^{**}$ and $\lceil e^* \rfloor^{\tau_s}_{@u} \simeq^C_\ell \lceil e^{**} \rfloor^{\tau_s}_{@u}$, hence $e^* = e^{**}$. By rule (O-LetCtx) we get

$$\begin{array}{lcl} Q & = & \langle K^*, N^*, H^*, \lceil \mathsf{let}\ x = e^*\ \mathsf{in}\ e_2 \rfloor^{\tau_s}_{@u}, [\,]\rangle \\ Q' & = & \langle K^{**}, N^{**}, H^{**}, \lceil \mathsf{let}\ x = e^*\ \mathsf{in}\ e_2 \rfloor^{\tau_s}_{@u}, [\,]\rangle \end{array}$$

and $o = o^*$ and $o' = o^{**}$ and the claim follows.

- Otherwise we know by the induction hypothesis that both P and P' use rule (O-Complete). We get

$$\begin{array}{lcl} Q & = & \langle K, N, H, \mathsf{wait}, [\,]\rangle \\ Q' & = & \langle K', N', H', \mathsf{wait}, [\,]\rangle \end{array}$$

and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.

* If $e = \mathsf{let}\ x = v\ \mathsf{in}\ e_1$ then (O-Let) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,]\rangle \\ Q' & = & \langle K', N', H', \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,]\rangle \end{array}$$

and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.

* If $e = \mathsf{get\text{-}ck}(k)$ then we distinguish two cases:
  - If there exist τ, v with $\mathsf{ck}(k, v)^\tau \in\, !K(u)^{\tau_s}$ then we know that $\mathsf{ck}(k, v)^\tau \in\, !K'(u)^{\tau_s}$ by Lemma 3. Hence rule (O-GetCookie) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \lceil v \rfloor^{\tau_s}_{@u}, [\,]\rangle \\ Q' & = & \langle K', N', H', \lceil v \rfloor^{\tau_s}_{@u}, [\,]\rangle \end{array}$$

    and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.
  - If there exist no τ, v with $\mathsf{ck}(k, v)^\tau \in\, !K(u)^{\tau_s}$ then we know that there are no τ', v' with $\mathsf{ck}(k, v')^{\tau'} \in\, !K(u)^{\tau_s}$ by Lemma 3. Hence rule (O-Complete) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \mathsf{wait}, [\,]\rangle \\ Q' & = & \langle K', N', H', \mathsf{wait}, [\,]\rangle \end{array}$$

    and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.

* If $e = \mathsf{set\text{-}ck}(k, v)$ then rule (O-SetCookie) is used. Let $CK = \{\mathsf{ck}(k, v)^\tau \mid \tau = \kappa(\mathit{host}(u), k)\}$, then

$$\begin{array}{lcl} Q & = & \langle K \xleftarrow{CK}_{\tau_s} u, N, H, \mathsf{wait}, [\,]\rangle \\ Q' & = & \langle K' \xleftarrow{CK}_{\tau_s} u, N', H', \mathsf{wait}, [\,]\rangle \end{array}$$

and $o = o' = \bullet$. We get $K \xleftarrow{CK}_{\tau_s} u \simeq^C_\ell K' \xleftarrow{CK}_{\tau_s} u$ using (1) and Lemma 4. The claim then follows using $(2), (3)$ and reflexivity.

∗ If $e = \mathsf{xhr}(u', \lambda x.e')$ then we distinguish two cases:
  · If *transfer*$(\mathsf{send}, \tau_s, tag(u')) = (\tau_n, -, \tau_{co}, -)$ for some τ_n, τ_{co} then rule (O-Xhr) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ Q' & = & \langle K', N', H' \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ o & = & \mathsf{xhr\_req}(u' : !K(u')^{\tau_{co}}) \\ o' & = & \mathsf{xhr\_req}(u' : !K'(u')^{\tau_{co}}) \end{array}$$

  We get $H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\} \simeq^C_\ell H' \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}$ using (3) and the definition of $\simeq^C_\ell$.
  We distinguish two subcases:
  (a) If $C(\tau_{co}) \cap C(\ell) \neq \emptyset$ then we get $!K(u')^{\tau_{co}} = !K'(u')^{\tau_{co}}$ and hence $o = o'$ by Lemma 3. The claim follows using (1), (2) and reflexivity.
  (b) If $C(\tau_{co}) \cap C(\ell) = \emptyset$ then we know $msg\_label(u) \subseteq C(l_{co})$ by the constraints of the transfer-function and can hence infer $msg\_label(u) \cap C(\ell) = \emptyset$. We then get $o \sim_\ell o'$ by the definition of $\sim_\ell$ and the claim follows using (1), (2) and reflexivity.
  · Otherwise rule (O-Complete) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \mathsf{wait}, [\,] \rangle \\ Q' & = & \langle K', N', H', \mathsf{wait}, [\,] \rangle \end{array}$$

  and $o = o' = \bullet$. The claim follows using (1), (2), (3) and reflexivity.
∗ For all other forms of e the rule (O-Complete) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \mathsf{wait}, [\,] \rangle \\ Q' & = & \langle K', N', H', \mathsf{wait}, [\,] \rangle \end{array}$$

and $o = o' = \bullet$. The claim follows using (1), (2), (3) and reflexivity.

□

A.2.3 Proof of Integrity

Definition 15 (High Equivalence)**.** *We define an* erasure *operator* $erase^I_\ell(\cdot)$ *on different browser data structures:*

- $erase^I_\ell(K)$ *is the map with domain* $dom(K)$ *such that for all* $d \in dom(K)$ *we have* $(erase^I_\ell(K))(d) = ck\text{-}erase^I_\ell(K(d))$
- $erase^I_\ell(N)$ *is the map obtained from* N *by erasing all the entries* $n^\tau : u$ *such that* $I(\tau) \cap I(\ell) \neq \emptyset$*;*
- $erase^I_\ell(H)$ *is the map obtained from* H *by erasing all the entries* $n^\tau : (u, \lceil \lambda x.e \rfloor_{@u'})$*' such that* $I(\tau) \cap I(\ell) \neq \emptyset$*;*

- *erase*$^I_\ell(T) =$ **wait** *whenever* $T = \lceil e \rfloor^\tau_{@u}$ *with* $I(\tau) \cap C(\ell) \neq \emptyset$, *while* *erase*$^I_\ell(T) = T$ *otherwise.*

We then define a binary high equivalence *relation* $\simeq^I_\ell$ *between data structures coinciding after applying the erasure* *erase*$^I_\ell(\cdot)$.

Definition 16 (Well-formedness)**.** *Let* $Q = \langle K, N, H, T, O \rangle$. *Then we say that* Q *is* well-formed*, if we have*

1. *if* $\{n^{\tau_n} : u\} \subseteq N$ *then* *msg_label*$(u) \subseteq I(\tau_n)$
2. *if* $\{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u'})\} \subseteq H$ *then* *msg_label*$(u) \subseteq I(\tau_n)$

Definition 17 (Candidate for Integrity Relation)**.** *Let* $Q = \langle K, N, H, T, O \rangle$ *and let* $Q' = \langle K', N', H', T', O' \rangle$, *we write* $Q \, \mathcal{R}^I_\ell \, Q'$ *if and only if: (1)* $K \simeq^I_\ell K'$; *(2)* $N \simeq^I_\ell N'$; *(3)* $H \simeq^I_\ell H'$; *(4)* $T \simeq^I_\ell T'$; *(5)* $O \approx_\ell O'$; *(6);* Q *and* Q' *are well-formed.*

Lemma 11. *For the initial state* C_0 *we have* $C_0 \, \mathcal{R}^I_\ell \, C_0$

Proof. (1) - (5) follow directly from the reflexivity of $\simeq^I_\ell$ and $\approx_\ell$. Well-formedness for the initial state is trivial since for $C_0 = \langle K, N, H, T, O \rangle$, we have $N = \{\}$ and $M = \{\}$. □

Lemma 12 (invariant)**.** *If* Q *is well-formed and* $Q \xrightarrow{a} Q'$ *then* Q' *is well-formed.*

Proof. Let $Q = \langle K, N, H, T, O \rangle$ and $Q' = \langle K', N', H', T', O' \rangle$. We do a case distinction on the form of Q:

- If Q is a consumer state then we do a case distinction on the used reduction rule.
  - If rule (I-Load) is used, Then we have $N' = N \uplus \{n^{\tau_n} : u\}$ with *msg_label*$(u) \subseteq I(\tau_n)$ by rule (T-Load), so 1) holds. Because of $H = H'$ and $T = T'$ we can also conclude 2).
  - If rule (I-DocResp) is used, then we have $N' \subseteq N'$ and $H' = H$ and we can conclude 1) and 2).
  - If rule (I-DocRedir) is used, then we have $N' = (N \setminus \{n^{\tau_n} : u\}) \uplus \{n^{\tau_m} : u'\}$ with *msg_label*$(u') \subseteq I(\tau_m)$ by rule (T-DocRedir), so 1) follows from 1). Because of $H = H'$ we can also conclude 2).
  - If rule (I-XhrResp) is used, then we $H' \subseteq H$ and $N' = N$ and we can conclude 1) and 2)
  - If rule (I-XhrRedir) is used, then we have $H' = (H \setminus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u''})\}) \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\}$ with *msg_label*$(u') \subseteq I(\tau_m)$ by rule (T-XhrRedir) and 2) follows from 2). Because of $N = N'$ we can also conclude 1).
  - If rule (I-Complete) is used, then we have $N' = N$ and $H' = H$, so 1) and 2) hold.

- If Q is a producer sate, then we have $O \neq [\,]$ or $T \neq \mathsf{wait}$. We perform a case distinction on these two possibilities.

  - If $O = o :: O''$ then rule (O-Flush) is used and $N = N'$ and $H = H'$, so 1) and 2) hold.
  - if $O = [\,]$ and $T = \lceil e \rfloor^{\tau}_{@u}$ then we do an induction on the term structure of e.
    Induction Hypothesis: The claim holds for all $Q = \langle K, N, H, \lceil e' \rfloor^{\tau}_{@u}, O\rangle$ where e' is a subterm of e.
    - If $e = \lambda x.e'\, v$ then (O-App) and $N' = N$ and $H = H'$. The claim follows trivially.
    - If $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ then we distinguish two cases:
      - If there exist K^*, N^*, H^*, e^* and o^* such that $\langle K, N, H, \lceil e_1 \rfloor^{\tau_s}_{@u}, [\,]\rangle \overset{o^*}{\longmapsto} \langle K^*, N^*, H^*, \lceil e^* \rfloor^{\tau_s}_{@u}, [\,]\rangle$ then (O-LetCtx) is used and $N' = N^*$ and $H' = H^*$. We get the claim by the induction hypothesis.
      - Otherwise rule (O-Complete) is used and the claim follows trivially.
    - If $e = \mathsf{let}\ x = v\ \mathsf{in}\ e'$ then (O-Let) is used and $N' = N$ and $H = H'$. The claim follows trivially.
    - If $e = \mathsf{get\text{-}ck}(k)$ then we distinguish two cases:
      - If there exist τ, v with $\mathsf{ck}(k, v)^{\tau} \in\, !K(u)^{\tau_s}$ then rule (O-GetCookie) is used and $N' = N$ and $H = H'$. The claim follows trivially.
      - Otherwise rule (O-Complete) is used and the claim follows trivially.
    - If $e = \mathsf{set\text{-}ck}(k, v)$ then rule (O-SetCookie) is used and $N' = N$ and $H = H'$. The claim follows trivially.
    - If $e = \mathsf{xhr}(u', \lambda x.e')$ then we distinguish two cases:
      - If $\mathit{transfer}(\mathsf{send}, \tau_s, \mathit{tag}(u')) = (\tau_n, -, \tau_{co}, -)$ for some τ_n, τ_{co} then rule (O-Xhr) is used and $H' = H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}$
        By the constraints we get $\mathit{msg\_label}(u') \subseteq I(\tau_n)$ and we get 2). Because of $N' = N$ we also have 1)
      - Otherwise rule (O-Complete) is used and the claim follows trivially.
    - For all other forms of e (O-Complete) is used and the claim follows trivially.

□

Lemma 13. *$\mathcal{R}^I_\ell$ satisfies the first condition of Definition 12.*

Proof. A straightforward syntactic check on the definition of $\mathcal{R}^I_\ell$. □

Lemma 14 (Equality of high-integrity cookies)**.** *Let $K \simeq^I_\ell K'$. Then $\mathit{ck\text{-}erase}^I_\ell(!K(u)^{\tau}) = \mathit{ck\text{-}erase}^I_\ell(!K'(u)^{\tau})$ for all u, τ.*

Proof. We only show $\textit{ck-erase}^I_\ell(!K(u)^\tau) \subseteq \textit{ck-erase}^I_\ell(!K'(u)^\tau)$, the proof of the other direction is analogous.

Let $\mathsf{ck}(k,v)^{\tau'} \in \textit{ck-erase}^I_\ell(!K(u)^\tau)$. Then we know that $\mathsf{ck}(k,v)^{\tau'} \in K(d)$ with $I(\tau') \cap I(\ell) = \emptyset$ and $\textit{transfer}(\mathsf{get}, \tau, \tau')\downarrow$. By the definition of $\simeq^I_\ell$ we know that $\mathsf{ck}(k,v)^{\tau'} \in K'(d)$ and because of $I(\tau') \cap I(\ell) = \emptyset$ and $\textit{transfer}(\mathsf{get}, \tau, \tau')\downarrow$ we know $\mathsf{ck}(k,v)^{\tau} \in \textit{ck-erase}^I_\ell(!K'(u)^{\tau'})$. □

Lemma 15 (High Equivalence after Cookie Updates)**.** *Let* $\Gamma \vdash$ *transfer and* $K \simeq^I_\ell K'$*. Then we have* $K \xleftarrow{CK}_\tau u \simeq^I_\ell K' \xleftarrow{CK}_\tau u$ *for all* CK, u, τ*.*

Proof. We have to show $\textit{erase}^I_\ell((K \xleftarrow{CK}_\tau u)(d)) = \textit{erase}^I_\ell((K' \xleftarrow{CK}_\tau u)(d))$ for all domains d. We only show $\textit{erase}^I_\ell((K \xleftarrow{CK}_\tau u)(d)) \subseteq \textit{erase}^I_\ell((K' \xleftarrow{CK}_\tau u)(d))$, the proof for the other direction is analogous. We distinguish two cases:

- If $d \neq \textit{host}(u)$ then $(K \xleftarrow{CK}_\tau u)(d) = K(d)$ and $(K' \xleftarrow{CK}_\tau u)(d) = K'(d)$ and the claim follows directly from $K \simeq^I_\ell K'$.
- If $d = \textit{host}(u)$ then assume that $\mathsf{ck}(k_1,v_1)^{\tau_1} \in \textit{erase}^I_\ell((K \xleftarrow{CK}_\tau u)(d))$. This implies that $I(\tau_1) \cap I(\ell) = \emptyset$ and $\mathsf{ck}(k_1,v_1)^{\tau_1} \in K \xleftarrow{CK}_\tau u$. We do a case distinction following the definition of $K \xleftarrow{CK}_\tau u$.
  - If $\mathsf{ck}(k_1,v_1)^{\tau_1} \in CK$ and $\textit{transfer}(\mathsf{set}, \tau, \tau_1)\downarrow$ then we can immediately conclude that $\mathsf{ck}(k_1,v_1)^{\tau_1} \in CK'$ and hence $\mathsf{ck}(k_1,v_1)^{\tau_1} \in \textit{erase}^I_\ell((K' \xleftarrow{CK}_\tau u)(d))$.
  - If $\mathsf{ck}(k_1,v_1)^{\tau_1} \in K(d)$ and if for all $\mathsf{ck}(k_2,v_2)^{\tau_2} \in CK$ we have $k_2 \neq k_1$ or $\textit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$ then let $\mathsf{ck}(k_3,v_3)^{\tau_3} \in CK'$. We have to show $k_3 \neq k_1$ or $\textit{transfer}(\mathsf{set}, \tau, \tau_3)\uparrow$. We distinguish two cases:
    - If $k_3 \neq k_1$ then the claim follows trivially.
    - if $k_3 = k_1$ then we observe that $\tau_3 = \kappa(d, k_3) = \kappa(d, k_1) = \tau_1$. This implies $I(\tau_3) \cap I(\ell) = \emptyset$ and hence $\mathsf{ck}(k_3,v_3)^{\tau_3} \in CK$ because of $\textit{ck-erase}^I_\ell(CK) = \textit{ck-erase}^I_\ell(CK')$. We then know $\textit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$.

□

Lemma 16 (Low Integrity Updates)**.** *Let* $\Gamma \vdash$ *transfer and* $I(\tau) \cap I(\ell) \neq \emptyset$*. Then we have* $K \simeq^I_\ell K \xleftarrow{CK}_\tau u$ *for all* CK, u*.*

Proof. We show both directions of set inclusions:

- Assume that $\mathsf{ck}(k,v)^{\tau'} \in \textit{erase}^I_\ell((K \xleftarrow{CK}_\tau u)(d))$. This implies that $I(\tau') \cap I(\ell) = \emptyset$ and $\mathsf{ck}(k,v)^{\tau'} \in (K \xleftarrow{CK}_\tau u)(d)$. We can deduce that $I(\tau) \not\subseteq I(\tau')$, hence we know $\textit{transfer}(\mathsf{set}, \tau, \tau')\uparrow$. By the definition of $(K \xleftarrow{CK}_\tau u)(d)$ we then know that $\mathsf{ck}(k,v)^{\tau'} \in K(d)$. Because of $I(\tau') \cap I(\ell) = \emptyset$ we get $\mathsf{ck}(k,v)^{\tau'} \in \textit{erase}^I_\ell(K(d))$.

- Assume that $\mathsf{ck}(k_1, v_1)^{\tau_1} \in \mathit{erase}^I_\ell(K(d))$. This implies that $I(\tau_1) \cap I(\ell) = \emptyset$ and $\mathsf{ck}(k_1, v_1)^{\tau_1} \in K(d)$. To show that $\mathsf{ck}(k_1, v_1)^{\tau_1} \in \mathit{erase}^I_\ell((K \xleftarrow{CK}_\tau u)(d))$ it suffices to show that for all we have $\mathsf{ck}(k_2, v_2)^{\tau_2} \in CK\ k_1 \neq k_2$ or $\mathit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$. We distinguish two cases:
  - If $k_2 \neq k_1$ then the claim follows trivially
  - If $k_2 = k_1$ then we observe that $\tau_2 = \kappa(\mathit{host}(u), k_2) = \kappa(\mathit{host}(u), k_1) = \tau_1$. As we know $I(\tau_1) \not\subseteq I(\tau)$, we can conclude $\mathit{transfer}(\mathsf{set}, \tau, \tau_2)\uparrow$

□

Lemma 17. *$\mathcal{R}^I_\ell$ satisfies the second condition of Definition 12.*

Proof. Let $C = \langle K, N, H, \mathsf{wait}, [\,]\rangle$ and $C' = \langle K', N', H', \mathsf{wait}, [\,]\rangle$ with $C\ \mathcal{R}^I_\ell\ C'$. By definition of $\mathcal{R}^C_\ell$, we have:

(1) $K \simeq^I_\ell K'$;

(2) $N \simeq^I_\ell N'$;

(3) $H \simeq^I_\ell H'$;

(4) C and C' are well-formed.

Assume that $C \xrightarrow{i} P$ and $C' \xrightarrow{i'} P'$ with $i \sim_\ell i'$ and $\mathit{rel}_\ell(i)$ and $\mathit{rel}_\ell(i')$. Well-formedness of P and P' follows directly from Lemma 12. We perform a case distinction on i:

- if $i = \mathsf{load}(u)$, then $i' = \mathsf{load}(u)$ by definition of $\sim_\ell$. By the reduction rules, assuming that $\mathit{transfer}(\mathsf{load}, \mathit{tag}(u), -) = (\ell_n, -, \ell_{co}, -)$, we then have:

$$\begin{array}{rcl} P & = & \langle K, N \uplus \{n^{\ell_n} : u\}, H, \mathsf{wait}, \mathsf{doc\_req}(u : !K(u)^{\ell_{co}})\rangle \\ P' & = & \langle K', N' \uplus \{n^{\ell_n} : u\}, H', \mathsf{wait}, \mathsf{doc\_req}(u : !K'(u)^{\ell_{co}})\rangle \end{array}$$

  To prove the desired conclusion, we need to show:

  (a) $N \uplus \{n^{\ell_n} : u\} \simeq^C_\ell N' \uplus \{n^{\ell_n} : u\}$

  (b) $\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}) \approx_\ell \mathsf{doc\_req}(u : !K'(u)^{\ell_{co}})$

  Point (a) is an immediate consequence of (2), while point (b) is more complicated. We distinguish two sub-cases:

  - if $\Gamma_I(u) \cap I(\ell) \neq \emptyset$ then we have $\neg\mathit{rel}_\ell(\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}))$ and we also have $\neg\mathit{rel}_\ell(\mathsf{doc\_req}(u : !K'(u)^{\ell_{co}}))$, hence we conclude by applying rules (S-Left) and (S-Right);

– if $\Gamma_I(u) \cap I(\ell) = \emptyset$ then we get $\mathit{ck\text{-}erase}^I_\ell(!K(u)^{\tau_{co}}) = \mathit{ck\text{-}erase}^I_\ell(!K'(u)^{\tau_{co}})$ by Lemma 14 and hence $\mathsf{doc\_req}(u : !K(u)^{\ell_{co}}) \sim_\ell \mathsf{doc\_req}(u : !K'(u)^{\ell_{co}})$ by the definition of $\sim_\ell$.

- if $i = \mathsf{doc\_resp}_n(u : CK, e)$, then $i' = i$ with $\mathit{msg\_label}(u) \not\subseteq I(\ell)$ by the definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:

  – assume that rule (I-DocResp) is used on both C and C', then we know that $N = N_0 \uplus \{n^{\tau_n} : u\}$ and $N' = N_1 \uplus \{n^{\tau'_n} : u\}$ for some N_0, N_1 such that $N_0 \simeq^I_\ell N_1$. Assuming that we have

    * $\mathit{transfer}(\mathsf{doc\_resp}, \tau_n, \mathit{tag}(u)) = (-, \tau_{ci}, -, \tau_s)$ and
    * $\mathit{transfer}(\mathsf{doc\_resp}, \tau'_n, \mathit{tag}(u)) = (-, \tau'_{ci}, -, \tau'_s)$

    we then get

    $$\begin{aligned} P &= \langle K \xleftarrow{CK}_{\tau_{ci}} u, N_0, H, \lceil e \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ P' &= \langle K' \xleftarrow{CK}_{\tau'_{ci}} u, N_1, H', \lceil e \rfloor^{\tau'_s}_{@u}, [\,] \rangle \end{aligned}$$

    To prove the desired conclusion, we need to show:

    (a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq^I_\ell K' \xleftarrow{CK}_{\tau'_{ci}} u$.

    (b) $\lceil e \rfloor^{\tau_s}_{@u} \simeq^I_\ell \lceil e \rfloor^{\tau'_s}_{@u}$

    We distinguish two subcases:

    * if $I(\tau_n) \cap I(\ell) = \emptyset$ then we know by (2) and the definition of $\simeq^I_\ell$ that $\tau_n = \tau'_n$. Hence we also have $\tau_{ci} = \tau'_{ci}$ and $\tau_s = \tau'_s$. Point (a) follows directly from Lemma 15 and point (b) follows from the definition of $\simeq^I_\ell$.
    * if $I(\tau_n) \cap I(\ell) \neq \emptyset$ then we know by (2) also that $I(\tau'_n) \cap I(\ell) \neq \emptyset$. By the constraints that $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau' ci, \tau_s, \tau'_s\}$. Then (a) follows from Lemma 16, (1) and transitivity and (b) follows directly from the definition of $\simeq^I_\ell$.

  – assume that rule (I-DocResp) is used on C, while C' fails using rule (I-Complete). This implies that $N = N_0 \uplus \{n^{\tau_n} : u\}$ for some N_0 with $N_0 \simeq^I_\ell N'$ and $I(\tau_n) \cap I(\ell) \neq \emptyset$. Assuming that $\mathit{transfer}(\mathsf{doc\_resp}, \tau_n, \mathit{tag}(u)) = (-, \tau_{ci}, -, \tau_s)$, we have:

    $$\begin{aligned} P &= \langle K \xleftarrow{CK}_{\tau_{ci}} u, N_0, H, \lceil e \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ P' &= \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{aligned}$$

    To prove the desired conclusion we need to show:

    (a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq^I_\ell K'$

    (b) $\lceil e \rfloor^{\tau_s}_{@u} \simeq^I_\ell \mathsf{wait}$

    (c) $[\,] \approx_\ell \bullet$

    We know by the constraints that $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau_s\}$. Then (a) follows from Lemma 16, (1) and transitivity and (b) follows directly from the definition of $\simeq^I_\ell$. Point (c) follows from $\neg \mathit{rel}(\bullet)$ and using rules (S-Right) and (S-Empty).

– assume that rule (I-DocResp) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

– assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{rcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

- if $i = \mathsf{doc\_redir}_n(u : CK, u')$, then $i' = i$ with $\mathit{msg\_label}(u) \not\subseteq I(\ell)$ by definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:

  – assume that rule (I-DocRedir) is used on both C and C', then we know that $N = N_0 \uplus \{n^{\tau_n} : u\}$ and $N' = N_1 \uplus \{n^{\tau'_n} : u\}$ for some N_0, N_1 such that $N_0 \simeq^I_\ell N_1$. Assuming that we have

  * $\mathit{transfer}(\mathsf{doc\_redir}, \tau_n, \mathit{tag}(u'), -) = (\tau_m, \tau_{ci}, \tau_{co}, -)$
  * $\mathit{transfer}(\mathsf{doc\_redir}, \tau'_n, \mathit{tag}(u')) = (\tau'_m, \tau'_{ci}, \tau'_{co}, -)$,
  * $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$ and
  * $K_1 = K' \xleftarrow{CK}_{\tau'_{ci}} u$

  then we get

$$\begin{array}{rcl} P & = & \langle K_0, N_0 \uplus \{n^{\tau_m} : u'\}, H, \mathsf{wait}, \mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \rangle \\ P' & = & \langle K_1, N_1 \uplus \{n^{\tau'_m} : u'\}, H', \mathsf{wait}, \mathsf{doc\_req}(u' : !K_1(u')^{\tau'_{co}}) \rangle \end{array}$$

  To prove the desired conclusion, we need to show:

  (a) $K_0 \simeq^I_\ell K_1$

  (b) $N_0 \uplus \{n^{\tau_m} : u'\} \simeq^I_\ell N_1 \uplus \{n^{\tau'_m} : u'\}$

  (c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell \mathsf{doc\_req}(u' : !K_1(u')^{\tau'_{co}})$

  We distinguish two subcases:

  * if $I(\tau_n) \cap I(\ell) = \emptyset$ then we know by (2) and the definition of $\simeq^I_\ell$ that $\tau_n = \tau'_n$. Hence we also have $\tau_m = \tau'_m$, $\tau_{ci} = \tau'_{ci}$ and $\tau_{co} = \tau'_{co}$. Point (a) follows directly from Lemma 15 and point (b) follows from (2) and the definition of $\simeq^I_\ell$. Using (a) and Lemma 14 we can conclude (c) using the definition of $\sim_\ell$ and rule (S-Match).
  * if $I(\tau_n) \cap I(\ell) \neq \emptyset$ then by (2) we know that $I(\tau'_n) \cap I(\ell) \neq \emptyset$. By the constraints of the transfer-function we get $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau'_{ci}, \tau_m, \tau'_m\}$ and $\Gamma_I(u') \cap I(\ell) \neq \emptyset$. Point (a) follows from Lemma 16, (1) and transitivity and point (b) follows from (2) and the definition of $\simeq^I_\ell$. Because of $\Gamma_I(u') \cap I(\ell) \neq \emptyset$ we get $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_1(u')^{\tau'_{co}}))$. We can conclude (c) by applying rules (S-Left) and (S-Right).

– assume that rule (I-DocRedir) is used on C, while C' fails using rule (I-Complete). This implies that $N = N_0 \uplus \{n^{\tau_n} : u\}$ for some N_0 with $N_0 \simeq^I_\ell N'$ and $I(\tau_n) \cap I(\ell) \neq \emptyset$. Assuming that *transfer*(doc_redir, τ_n, *tag*(u')) = $(\tau_m, \tau_{ci}, \tau_{co}, -)$ and $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$ we have:

$$\begin{array}{lcl} P & = & \langle K_0, N_0 \uplus \{n^{\tau_m} : u'\}, H, \mathsf{wait}, \mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}})\rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet\rangle \end{array}$$

To prove the desired conclusion we need to show:

(a) $K_0 \simeq^I_\ell K'$

(b) $N_0 \uplus \{n^{\tau_m} : u'\} \simeq^I_\ell N'$

(c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell \bullet$

By the constraints of the transfer-function we get $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau_s\}$ and $\Gamma_I(u') \cap I(\ell) \neq \emptyset$.. Point (a) follows from Lemma 16, (1) and transitivity and point (b) follows from (2) and the definition of $\simeq^I_\ell$. Because of $\Gamma_I(u') \cap I(\ell) \neq \emptyset$ we get $\neg$*rel*($\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}})$) and $\neg$*rel*($\bullet$). We can conclude (c) by applying rules (S-Left) and (S-Right).

– assume that rule (I-DocRedir) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

– assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{lcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet\rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet\rangle \end{array}$$

The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

• if $i = \mathsf{xhr\_resp}_n(u : CK, v)$, then $i' = i$ with *msg_label*(u) $\not\subseteq I(\ell)$ by definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:

– assume that rule (I-XhrResp) is used on both C and C', then we know that $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u'})\}$ and $H' = H_1 \uplus \{n^{\tau'_n} : (u, \lceil \lambda x.e' \rfloor_{@u''})\}$ for some H_0, H_1 such that $H_0 \simeq^I_\ell H_1$. Assuming that *transfer*(xhr_resp, τ_n, *tag*(u)) = $(-, \tau_{ci}, -, \tau_s)$ and *transfer*(xhr_resp, τ'_n, *tag*(u)) = $(-, \tau'_{ci}, -, \tau'_s)$, we have:

$$\begin{array}{lcl} P & = & \langle K \xleftarrow{CK}_{\tau_{ci}} u, N, H_0, \lceil e\{v/x\} \rfloor^{\tau_s}_{@u'}, [\,]\rangle \\ P' & = & \langle K' \xleftarrow{CK}_{\tau'_{ci}} u, N', H_1, \lceil e\{v/x\} \rfloor^{\tau'_s}_{@u''}, [\,]\rangle \end{array}$$

To prove the desired conclusion, we need to show:

(a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq^I_\ell K' \xleftarrow{CK}_{\tau'_{ci}} u$.

(b) $\lceil e\{v/x\} \rfloor^{\tau_s}_{@u'} \simeq^I_\ell \lceil e'\{v/x\} \rfloor^{\tau'_s}_{@u''}$

We distinguish two subcases:

  - if $I(\tau_n) \cap I(\ell) = \emptyset$ then we know by (3) and the definition of $\simeq^I_\ell$ that $\tau_n = \tau'_n$, $u' = u''$ and $e = e'$. Hence we also have $\tau_{ci} = \tau'_{ci}$ and $\tau_s = \tau'_s$. Point (a) follows directly from Lemma 15 and point (b) follows from the definition of $\simeq^I_\ell$.
  - if $I(\tau_n) \cap I(\ell) \neq \emptyset$ then we know by (3) that $I(\tau'_n) \cap I(\ell) \neq \emptyset$. By the constraints we get $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau'_{ci}, \tau_s, \tau'_s\}$. Point (a) then follows from Lemma 16, (1) and transitivity and point (b) follows from the definition of $\simeq^I_\ell$.

- assume that rule (I-XhrResp) is used on C, while C' fails using rule (I-Complete). This implies that $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u'})\}$ for some H_0 with $H_0 \simeq^I_\ell H'$ and $I(\tau_n) \cap I(\ell) \neq \emptyset$. Assuming that $\mathit{transfer}(\mathsf{xhr\_resp}, \tau_n, -) = (-, \tau_{ci}, -, \tau_s)$, we have:

$$\begin{array}{rcl} P & = & \langle K \xleftarrow{CK}_{\tau_{ci}} u, N, H_0, \lceil e\{v/x\} \rfloor^{\tau_s}_{@u'}, [\,] \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

  To prove the desired conclusion we need to show:

  (a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq^I_\ell K'$
  (b) $\lceil e\{v/x\} \rfloor^{\tau_s}_{@u'} \simeq^I_\ell \mathsf{wait}$
  (c) $[\,] \approx_\ell \bullet$

  we know $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau_s\}$ by the constraints. Point (a) then follows from Lemma 16, (1) and transitivity and point (b) follows from the definition of $\simeq^I_\ell$. Point (c) follows form $\neg \mathit{rel}(\bullet)$ and using rule (S-Right).

- assume that rule (I-XhrResp) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

- assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{rcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

  The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

- if $i = \mathsf{xhr\_redir}_n(u : CK, u')$, then $i' = i$ with $\mathit{msg\_label}(u) \not\subseteq I(\ell)$ by definition of $\sim_\ell$. We have four different sub-cases, based on the applied reduction rules:

  - assume that rule (I-XhrRedir) is used on both C and C', then we get $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u''})\}$ and $H' = H_1 \uplus \{n^{\tau'_n} : (u, \lceil \lambda x.e' \rfloor_{@u'''})\}$ for some H_0, H_1 such that $H_0 \simeq^I_\ell H_1$. Assuming that $\mathit{transfer}(\mathsf{xhr\_redir}, \tau_n, \mathit{tag}(u'), -) = (\tau_m, \tau_{ci}, \tau_{co}, -)$, $\mathit{transfer}(\mathsf{xhr\_redir}, \tau'_n, \mathit{tag}(u')) = (\tau'_m, \tau'_{ci}, \tau'_{co}, -)$, $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$ and $K_1 = K' \xleftarrow{CK'}_{\tau'_{ci}} u$, we have

$$\begin{array}{rcl} P & = & \langle K_0, N, H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}) \rangle \\ P' & = & \langle K_1, N', H_1 \uplus \{n^{\tau'_m} : (u', \lceil \lambda x.e' \rfloor_{@u'''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K_1(u')^{\tau'_{co}}) \rangle \end{array}$$

    To prove the desired conclusion, we need to show:

(a) $K_0 \simeq^I_\ell K_1$
(b) $H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\} \simeq^I_\ell H_1 \uplus \{n^{\tau'_m} : (u', \lceil \lambda x.e' \rfloor_{@u'''})\}$
(c) $\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}) \sim_\ell \mathsf{xhr\_req}(u' : !K_1(u')^{\tau'_{co}})$

We distinguish two subcases:

* if $I(\tau_n) \cap I(\ell) = \emptyset$ then we know by (3) and the definition of $\simeq^I_\ell$ that $\tau_n = \tau'_n$, $e = e'$ and $u'' = u'''$. Hence we also have $\tau_m = \tau'_m$, $\tau_{ci} = \tau'_{ci}$ and $\tau_{co} = \tau'_{co}$. Point (a) follows directly from Lemma 15 and point (b) follows from (3) and the definition of $\simeq^I_\ell$. Using (a) and Lemma 14 we can conclude (c) using the definition of $\sim_\ell$ and rule (S-Match).
* if $I(\tau_n) \cap I(\ell) \neq \emptyset$ then we know by (3) that also $I(\tau'_n) \cap I(\ell) \neq \emptyset$. By the constraints we get $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau'_{ci}, \tau_s, \tau'_s\}$ and $\Gamma_I(u') \cap I(\ell) \neq \emptyset$. Point (a) follows from Lemma 16, (1) and transitivity and point (b) follows directly from the definition of $\simeq^I_\ell$. Because of $\Gamma_I(u') \cap I(\ell) \neq \emptyset$ we get $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_1(u')^{\tau'_{co}}))$. We can conclude (c) by applying rules (S-Left) and (S-Right).

– assume that rule (I-XhrRedir) is used on C, while C' fails using rule (I-Complete). This implies that $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u''})\}$ for some H_0 with $H_0 \simeq^I_\ell H'$ and $I(\tau_n) \cap C(\ell) \neq \emptyset$. Assuming that $\mathit{transfer}(\mathsf{xhr\_redir}, \tau_n, \mathit{tag}(u')) = (\tau_m, \tau_{ci}, \tau_{co}, -)$ and $K_0 = K \xleftarrow[\tau_{ci}]{CK} u$ we have

$$\begin{array}{lcl} P & = & \langle K_0, N, H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}) \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

To prove the desired conclusion we need to show:

(a) $K_0 \simeq^I_\ell K'$
(b) $H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\} \simeq^I_\ell H'$
(c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell \bullet$

By the constraints we get $I(\tau) \cap I(\ell) \neq \emptyset$ for all $\tau \in \{\tau_{ci}, \tau_s\}$ and $\Gamma_I(u') \cap I(\ell) \neq \emptyset$. Point (a) follows from Lemma 16, (1) and transitivity and point (b) follows directly from the definition of $\simeq^I_\ell$. Because of $\Gamma_I(u') \cap I(\ell) \neq \emptyset$ we get $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}))$ and $\neg \mathit{rel}(\bullet)$. We can conclude (c) by applying rules (S-Left) and (S-Right).

– assume that rule (I-XhrRedir) is used on C', while C fails using rule (I-Complete). The case is analogous to the previous one;

– assume that rule (I-Complete) is used on both C and C', then:

$$\begin{array}{lcl} P & = & \langle K, N, H, \mathsf{wait}, \bullet \rangle \\ P' & = & \langle K', N', H', \mathsf{wait}, \bullet \rangle \end{array}$$

The conclusion is trivial, using (1), (2), (3) and the reflexivity of the stream similarity relation;

□

Lemma 18. *$\mathcal{R}_\ell^C$ satisfies the third condition of Definition 12.*

Proof. Let $C = \langle K, N, H, \mathsf{wait}, [\,]\rangle$ and $C' = \langle K', N', H', \mathsf{wait}, [\,]\rangle$ with $C\,\mathcal{R}_\ell^I\,C'$. By definition of $\mathcal{R}_\ell^C$, we have:

(1) $K \simeq_\ell^I K'$;

(2) $N \simeq_\ell^I N'$;

(3) $H \simeq_\ell^I H'$;

(4) C and C' are well-formed.

Assume that $C \xrightarrow{i} P$ with $\neg \mathit{rel}_\ell(i)$. Well-formedness of P follows directly from Lemma 12. We do a case distinction on the applied reduction rule:

- If rule (I-Load) is used then $i = \mathsf{load}(u)$. As we always have $\mathit{rel}(i)$ we do not have to consider this case.
- If rule (I-DocResp) is used then $i = \mathsf{doc\_resp}_n(u : CK, e)$. We have $\mathit{msg\_label}(u) \subseteq I(\ell)$ because of $\neg\mathit{rel}(i)$ and $N = N_0 \uplus \{n^{\tau_n} : u\}$ for some N_0 with $N_0 \simeq_\ell^I N'$. Assuming that $\mathit{transfer}(\mathsf{doc\_resp}, \tau_n, \mathit{tag}(u)) = (-, \tau_{ci}, -, \tau_s)$, we have:

$$P \;=\; \langle K \xleftarrow{CK}_{\tau_{ci}} u, N_0, H, \lceil e \rfloor^{\tau_s}_{@u}, [\,]\rangle$$

  We have to show:

  (a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq_\ell^I K'$

  (b) $\lceil e \rfloor^{\tau_s}_{@u} \simeq_\ell^I \mathsf{wait}$

  By well-formedness we know that $\mathit{msg\_label}(u) \subseteq I(\tau_n)$, hence $I(\tau_n) \cap I(\ell) \neq \emptyset$. By the constraints we can also derive $I(\tau_{ci}) \cap I(\ell) \neq \emptyset$ and $I(\tau_s) \cap I(\ell) \neq \emptyset$. Point (a) follows from (1) and Lemma 16 and point (b) follows from the definition of $\simeq_\ell^I$.

- If rule (I-DocRedir) is used, then $i = \mathsf{doc\_redir}_n(u : CK, u')$. We have $\mathit{msg\_label}(u) \subseteq I(\ell)$ because of $\neg\mathit{rel}(i)$ and $N = N_0 \uplus \{n^{\tau_n} : u\}$ for some N_0 with $N_0 \simeq_\ell^I N'$. Assuming that $\mathit{transfer}(\mathsf{doc\_redir}, \tau_n, \mathit{tag}(u')) = (\tau_m, \tau_{ci}, \tau_{co}, -)$ and $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$ we have:

$$P \;=\; \langle K_0, N_0 \uplus \{n^{\tau_m} : u'\}, H, \mathsf{wait}, \mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}})\rangle$$

  To prove the desired conclusion we need to show:

  (a) $K_0 \simeq_\ell^I K'$

(b) $N_0 \uplus \{n^{\tau_m} : u'\} \simeq^I_\ell N'$

(c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell [\,]$

By well-formedness we know that $\mathit{msg\_label}(u) \subseteq I(\tau_n)$, hence $I(\tau_n) \cap I(\ell) \neq \emptyset$. By the constraints of the transfer-function we also get $I(\ell_m) \cap I(\ell) \neq \emptyset$, $I(\ell_{ci}) \cap I(\ell) \neq \emptyset$ and $\Gamma_I(u') \cap I(\ell) \neq \emptyset$. Point (a) follows from Lemma 16 and point (b) follows from (2) and the definition of $\simeq^I_\ell$. By the definition of *rel* we get $\neg \mathit{rel}(\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}))$. We can conclude (c) by applying rules (S-Left) and (S-Empty).

- If rule (I-XhrResp) is used then $i = \mathsf{xhr\_resp}_n(u : CK, v)$. We have $\mathit{msg\_label}(u) \subseteq I(\ell)$ because of $\neg \mathit{rel}(i)$ and $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u'})\}$ for some H_0 with $H_0 \simeq^I_\ell H'$. Assuming that $\mathit{transfer}(\mathsf{xhr\_resp}, \tau_n, -) = (-, \tau_{ci}, -, \tau_s)$, we have:

$$P \quad = \quad \langle K \xleftarrow{CK}_{\tau_{ci}} u, N, H_0, \lceil e\{v/x\} \rfloor^{\tau_s}_{@u'}, [\,] \rangle$$

To prove the desired conclusion we need to show:

(a) $K \xleftarrow{CK}_{\tau_{ci}} u \simeq^I_\ell K'$

(b) $\lceil e\{v/x\} \rfloor^{\tau_s}_{@u'} \simeq^I_\ell \mathsf{wait}$

By well-formedness we know that $\mathit{msg\_label}(u) \subseteq I(\tau_n)$, hence $I(\tau_n) \cap I(\ell) \neq \emptyset$. By the constraints we can also derive $I(\tau_{ci}) \cap I(\ell) \neq \emptyset$ and $I(\tau_s) \cap I(\ell) \neq \emptyset$. Point (a) follows from (1) and Lemma 16 and point (b) follows from the definition of $\simeq^I_\ell$.

- If rule (I-XhrRedir) is used then $i = \mathsf{xhr\_redir}_n(u : CK, u')$, We have $\mathit{msg\_label}(u) \subseteq I(\ell)$ because of $\neg \mathit{rel}(i)$ and $H = H_0 \uplus \{n^{\tau_n} : (u, \lceil \lambda x.e \rfloor_{@u''})\}$ for some H_0 with $H_0 \simeq^I_\ell H'$ and $I(\tau_n) \cap C(\ell) \neq \emptyset$. Assuming that $\mathit{transfer}(\mathsf{xhr\_redir}, \tau_n, \mathit{tag}(u')) = (\tau_m, \tau_{ci}, \tau_{co}, -)$ and $K_0 = K \xleftarrow{CK}_{\tau_{ci}} u$ we have

$$P \quad = \quad \langle K_0, N, H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\}, \mathsf{wait}, \mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}) \rangle$$

To prove the desired conclusion we need to show:

(a) $K_0 \simeq^I_\ell K'$

(b) $H_0 \uplus \{n^{\tau_m} : (u', \lceil \lambda x.e \rfloor_{@u''})\} \simeq^I_\ell H'$

(c) $\mathsf{doc\_req}(u' : !K_0(u')^{\tau_{co}}) \approx_\ell [\,]$

By well-formedness we know that $\mathit{msg\_label}(u) \subseteq I(\tau_n)$, hence $I(\tau_n) \cap I(\ell) \neq \emptyset$. By the constraints of the transfer-function we also get $I(\ell_m) \cap I(\ell) \neq \emptyset$, $I(\ell_{ci}) \cap I(\ell) \neq \emptyset$ and $\Gamma_I(u') \cap I(\ell) \neq \emptyset$. Point (a) follows from Lemma 16 and point (b) follows from (3) and the definition of $\simeq^I_\ell$. By the definition of *rel* we get $\neg \mathit{rel}(\mathsf{xhr\_req}(u' : !K_0(u')^{\tau_{co}}))$. We can conclude (c) by applying rules (S-Left) and (S-Empty).

- if rule (I-Complete) is used, then we have

$$P \quad = \quad \langle K, N, H, \mathsf{wait}, \bullet \rangle$$

and the claim follows trivially.

□

Lemma 19 (Low Integrity Scripts)**.** *Let* $P = \langle K, N, H, \lceil e \rfloor^{\tau_s}_{@u}, [\,] \rangle$ *and let* $P \, \mathcal{R}^I_\ell \, Q$ *for some state* Q*. If* $I(\tau_s) \cap I(\ell) \neq \emptyset$ *and* $P \xrightarrow{o} Q'$ *for some* o, Q'*, then* $\neg rel_\ell(o)$ *and* $Q' \, \mathcal{R}^I_\ell \, Q$*.*

Proof. Let $Q = \langle K', N', H', T', O' \rangle$. By definition of $\mathcal{R}^I_\ell$ we have

(1) $K \simeq^I_\ell K'$;

(2) $N \simeq^I_\ell N'$;

(3) $H \simeq^I_\ell H'$;

(4) $\lceil e \rfloor^{\tau_s}_{@u} \simeq^I_\ell T'$;

(5) $[\,] \simeq^I_\ell O'$;

(6) P and Q are well-formed.

Well-formedness of Q' follows directly from Lemma 12. We do an induction on the term structure of e:
Induction Hypothesis: The claim holds for all $P = \langle K, N, H, \lceil e' \rfloor^{\tau_s}_{@u}, [\,] \rangle$ where e' is a subterm of e.

In the following subcases we will say that the claim follows trivially if the claim follows directly from $(1) - (5)$ and the fact that

- If $e = \lambda x.e'\, v$ then (O-App) is used and $Q' = \langle K, N, H, \lceil e'\{v/x\} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. The claim follows trivially.
- If $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ then we distinguish two cases:
  - If there exist K^*, N^*, H^*, e^* and o^* such that we have $\langle K, N, H, \lceil e_1 \rfloor^{\tau_s}_{@u}, [\,] \rangle \xmapsto{o^*} \langle K^*, N^*, H^*, \lceil e^* \rfloor^{\tau_s}_{@u}, [\,] \rangle$ then (O-LetCtx) is used and $Q' = \langle K^*, N^*, H^*, \lceil \mathsf{let}\ x = e^*\ \mathsf{in}\ e_2 \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = o^*$. As e_1 is a subterm of e we get $K^* \simeq^I_\ell K'$, $N^* \simeq^I_\ell N'$, $H^* \simeq^I_\ell H'$ and $\neg rel(o^*)$ by the induction hypothesis. The claim then follows trivially.
  - Otherwise rule (O-Complete) is used and the claim follows trivially.
- If $e = \mathsf{let}\ x = v\ \mathsf{in}\ e'$ then (O-Let) is used and $Q' = \langle K, N, H, \lceil e'\{v/x\} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. The claim follows trivially.
- If $e = \mathsf{get\text{-}ck}(k)$ then we distinguish two cases:
  - If there exist τ, v with $\mathsf{ck}(k, v)^\tau \in\, !K(u)^{\tau_s}$ then rule (O-GetCookie) is used and $Q' = \langle K, N, H, \lceil v \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. The claim follows trivially.
  - Otherwise rule (O-Complete) is used and the claim follows trivially.

- If $e = \mathsf{set\text{-}ck}(k, v)$ then rule (O-SetCookie) is used. Let $CK = \{\mathsf{ck}(k, v)^{\tau} \mid \tau = \kappa(\mathit{host}(u), k)\}$, then , then $Q' = \langle K \xleftarrow{CK}_{\tau_s} u, N, H, \lceil \mathit{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and $o = \bullet$. By Lemma 16 and (1) we get $K \xleftarrow{CK}_{\tau_s} u \simeq^I_\ell K'$ The claim then follows trivially.
- If $e = \mathsf{xhr}(u', \lambda x.e')$ then we distinguish two cases:
  - If $\mathit{transfer}(\mathsf{send}, \tau_s, \mathit{tag}(u')) = (\tau_n, -, \tau_{co}, -)$ for some τ_n, τ_{co} then rule (O-Xhr) is used and $Q' = \langle K, N, H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle$ and we have $o = \mathsf{xhr\_req}(u' : !K(u')^{\tau_{co}})$.
    By the constraints we get $I(\tau_n) \cap I(\ell) \neq \emptyset$ and we get $H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\} \simeq^I_\ell H'$ directly from the definition of $\simeq^I_\ell$ and (3). We also get $\Gamma_I(u') \cap I(\ell) \neq \emptyset$. so we can conclude $\neg \mathit{rel}(o)$. The claim then follows trivially.
  - Otherwise rule (O-Complete) is used and the claim follows trivially.
- For all other forms of e (O-Complete) is used and the claim follows trivially.

□

Lemma 20. *$\mathcal{R}^I_\ell$ satisfies the fourth condition of Definition 12.*

Proof. Let $P = \langle K, N, H, T, O \rangle$ and $C = \langle K', N', H', \mathsf{wait}, [\,] \rangle$, where either $T \neq \mathsf{wait}$ or $O = o :: O'$ for some o, O' by definition of producer state. Assume that $P \mathcal{R}^I_\ell C$, by definition we have:

(1) $K \simeq^I_\ell K'$;

(2) $N \simeq^I_\ell N'$;

(3) $H \simeq^I_\ell H'$;

(4) $T \simeq^I_\ell \mathsf{wait}$;

(5) $O \approx_\ell [\,]$;

(6) P and C are well-formed.

Let $P \xrightarrow{o} Q$. Well-formedness of Q follows directly from Lemma 12. We now distinguish two sub-cases:

- if $O = o :: O'$ for some o, O', the only available reduction rule for P is rule (O-Flush), hence:
  $$P \xrightarrow{o} \langle K, N, H, T, O' \rangle.$$
  Using (5) and the definition of stream similarity, we have $\neg \mathit{rel}_\ell(o)$ and $O' \approx_\ell [\,]$, which is enough to close the case;

- otherwise, assume without loss of generality that $O = [\,]$, then $T = \lceil e \rfloor^{\tau_s}_{@u}$ for some e, u, τ_s. We observe that point (4) implies that $I(\tau_s) \cap I(\ell) \neq \emptyset$, hence we conclude by Lemma 19;

□

Lemma 21. *$\mathcal{R}^I_\ell$ satisfies the fifth condition of Definition 12.*

Proof. Let $P = \langle K, N, H, T, O\rangle$ and $P' = \langle K', N', H', T', O'\rangle$ with $P\,\mathcal{R}^I_\ell\,P'$. By definition of $\mathcal{R}^I_\ell$, we have:

1. $K \simeq^I_\ell K'$;
2. $N \simeq^I_\ell N'$;
3. $H \simeq^I_\ell H'$;
4. $T \simeq^I_\ell T'$;
5. $O \approx_\ell O'$.
6. P and P' are well-formed.

Assume $P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$. Well-formedness of Q and Q' follows directly from Lemma 12. We distinguish four sub-cases:

- if $O = o :: O_0$ and $O' = o' :: O_1$, the only available reduction rule is (O-Flush) for both P and P'. We distinguish three sub-cases:
  - If $\neg \mathit{rel}(o)$ then $O \approx_\ell O'$ is shown by the rule (S-Left) and we know $O_0 \approx_\ell O'$. We have $P \xrightarrow{o} Q$ for $Q = \langle K, N, H, T, O_0\rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case b).
  - If $\mathit{rel}(o)$ and $\neg \mathit{rel}(o')$ then $O \approx_\ell O'$ is shown by the rule (S-Right) and we know $O \approx_\ell O_1$. We have $P' \xrightarrow{o} Q'$ for $Q' = \langle K', N', H', T', O_1\rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case c).
  - If $\mathit{rel}(o)$ and $\mathit{rel}(o')$ then $O \approx_\ell O'$ is shown by the rule (S-Match) and we know $O_0 \approx_\ell O_1$ and $o \sim_\ell o'$. By the rule (O-Flush) we have $P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ for $Q = \langle K, N, H, T, O_1\rangle$ and $Q' = \langle K', N', H', T', O_0\rangle$. Using $(1), (2), (3)$ and (4), we can match case a).
- if $O = o :: O''$ and $O' = [\,]$, we know that $O \approx_\ell O'$ can only be shown using rule (S-Left), hence we have $\neg \mathit{rel}(o)$ and $O'' \approx_\ell [\,]$. We have $P \xrightarrow{o} Q$ for $Q = \langle K, N, H, T, O''\rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case b).
- if $O = [\,]$ and $O' = o' :: O''$, we know that $O \approx_\ell O'$ can only be shown using rule (S-Right), hence we have $\neg \mathit{rel}(o')$ and $[\,] \approx_\ell O''$. We have $P' \xrightarrow{o'} Q'$ for $Q' = \langle K', N', H', T', O''\rangle$ by the rule (O-Flush). Using $(1), (2), (3)$ and (4), we can match case c).

- if $O = O' = [\,]$, then by definition of producer state we have $T = \lceil e \rfloor^{\tau_s}_{@u}$ and $T' = \lceil e' \rfloor^{\tau'_s}_{@u'}$. We distinguish two sub-cases:
  - If $I(\tau_s) \cap I(\ell) \neq \emptyset$ and $P \xrightarrow{o} Q$ then we get $\neg \mathit{rel}(o)$ and $Q \,\mathcal{R}^I_\ell\, P'$ by Lemma 19 and we can match case b).
  - If $I(\tau_s) \cap I(\ell) = \emptyset$ then point (4) implies $e = e'$, $u = u'$ and $\tau_s = \tau'_s$.
    We show the following stronger claim that directly implies a): $o \sim_\ell o'$ and $Q \,\mathcal{R}^I_\ell\, Q'$ and $P \overset{o}{\mapsto} Q \iff P \overset{o'}{\mapsto} Q'$
    We do an induction on the term structure of e:
    Induction Hypothesis: The claim holds for all $P = \langle K, N, H, \lceil e'' \rfloor^{\tau_s}_{@u}, O\rangle$ and $P' = \langle K', N', H', \lceil e'' \rfloor^{\tau_s}_{@u}, O'\rangle$ where e'' is a subterm of e.
    - If $e = \lambda x.e_1\ v$ then (O-App) is used on P and P' and
      $$\begin{array}{lcl} Q & = & \langle K, N, H, \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,]\rangle \\ Q' & = & \langle K', N', H', \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,]\rangle \end{array}$$
      and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.
    - If $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ then we distinguish two cases:
      - If there exist K^*, N^*, H^*, e^* and o^* such that $\langle K, N, H, \lceil e_1 \rfloor^{\tau_s}_{@u}, [\,]\rangle \overset{o^*}{\longmapsto} \langle K^*, N^*, H^*, \lceil e^* \rfloor^{\tau_s}_{@u}, [\,]\rangle$ then as e_1 is a subterm of e by the induction hypothesis we know that there exist $K^{**}, N^{**}, H^{**}, e^{**}$ and o^{**} such that $\langle K', N', H', \lceil e_1 \rfloor^{\tau_s}_{@u}, [\,]\rangle \overset{o^{**}}{\longmapsto} \langle K^{**}, N^{**}, H^{**}, \lceil e^{**} \rfloor^{\tau_s}_{@u}, [\,]\rangle$ such that $o^* \sim_\ell o^{**}$, $K^* \simeq^I_\ell K^{**}$, $N^* \simeq^I_\ell N^{**}$, $H^* \simeq^I_\ell H^{**}$ and $\lceil e^* \rfloor^{\tau_s}_{@u} \simeq^I_\ell \lceil e^{**} \rfloor^{\tau_s}_{@u}$, hence $e^* = e^{**}$. By rule (O-LetCtx) we get
        $$\begin{array}{lcl} Q & = & \langle K^*, N^*, H^*, \lceil \mathsf{let}\ x = e^*\ \mathsf{in}\ e_2 \rfloor^{\tau_s}_{@u}, [\,]\rangle \\ Q' & = & \langle K^{**}, N^{**}, H^{**}, \lceil \mathsf{let}\ x = e^*\ \mathsf{in}\ e_2 \rfloor^{\tau_s}_{@u}, [\,]\rangle \end{array}$$
        and $o = o^*$ and $o' = o^{**}$ and the claim follows.
      - Otherwise we know by the induction hypothesis that both P and P' use rule (O-Complete). We get
        $$\begin{array}{lcl} Q & = & \langle K, N, H, \mathsf{wait}, [\,]\rangle \\ Q' & = & \langle K', N', H', \mathsf{wait}, [\,]\rangle \end{array}$$
        and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.
    - If $e = \mathsf{let}\ x = v\ \mathsf{in}\ e_1$ then (O-Let) is used on P and P' and
      $$\begin{array}{lcl} Q & = & \langle K, N, H, \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,]\rangle \\ Q' & = & \langle K', N', H', \lceil e_1\{v/x\} \rfloor^{\tau_s}_{@u}, [\,]\rangle \end{array}$$
      and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.
    - If $e = \mathsf{get\text{-}ck}(k)$ then we distinguish two cases:

· If there exist τ, v with $\mathsf{ck}(k,v)^{\tau} \in !K(u)^{\tau_s}$ then we know by the constraints that $I(\tau) \cap I(\ell) = \emptyset$ and hence $\mathsf{ck}(k,v)^{\tau} \in \mathit{ck\text{-}erase}^I_\ell(!K(u)^{\tau_s})$. By Lemma 14 we know that $\mathsf{ck}(k,v)^{\tau} \in \mathit{ck\text{-}erase}^I_\ell(!K'(u)^{\tau_s})$ and hence also $\mathsf{ck}(k,v)^{\tau} \in !K'(u)^{\tau_s}$. Hence rule (O-GetCookie) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \lceil v \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ Q' & = & \langle K', N', H', \lceil v \rfloor^{\tau_s}_{@u}, [\,] \rangle \end{array}$$

and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.

· If there exist no τ, v with $\mathsf{ck}(k,v)^{\tau} \in !K(u)^{\tau_s}$ then we know that there are no τ', v' with $\mathsf{ck}(k,v')^{\tau'} \in !K(u)^{\tau_s}$ by Lemma 14. Hence rule (O-Complete) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \mathsf{wait}, [\,] \rangle \\ Q' & = & \langle K', N', H', \mathsf{wait}, [\,] \rangle \end{array}$$

and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.

∗ If $e = \mathsf{set\text{-}ck}(k,v)$ then rule (O-SetCookie) is used on .
Let $CK = \{\mathsf{ck}(k,v)^{\tau} \mid \tau = \kappa(\mathit{host}(u), k)\}$, then

$$\begin{array}{lcl} Q & = & \langle K \xleftarrow{CK}_{\tau_s} u, N, H, \mathsf{wait}, [\,] \rangle \\ Q' & = & \langle K' \xleftarrow{CK}_{\tau_s} u, N', H', \mathsf{wait}, [\,] \rangle \end{array}$$

and $o = o' = \bullet$. We get $K \xleftarrow{CK}_{\tau_s} u \simeq^I_\ell K' \xleftarrow{CK}_{\tau_s} u$ using (1) and Lemma 16. The claim then follows using $(2), (3)$ and reflexivity.

∗ If $e = \mathsf{xhr}(u', \lambda x.e')$ then we distinguish two cases:

· If $\mathit{transfer}(\mathsf{send}, \tau_s, \mathit{tag}(u')) = (\tau_n, -, \tau_{co}, -)$ for some τ_n, τ_{co} then rule (O-Xhr) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ Q' & = & \langle K', N', H' \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [\,] \rangle \\ o & = & \mathsf{xhr\_req}(u' : !K(u')^{\tau_{co}}) \\ o' & = & \mathsf{xhr\_req}(u' : !K'(u')^{\tau_{co}}) \end{array}$$

We get $H \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\} \simeq^I_\ell H' \uplus \{n^{\tau_n} : (u', \lceil \lambda x.e \rfloor_{@u})\}$ using (3) and the definition of $\simeq^I_\ell$. By Lemma 14 we get $\mathit{ck\text{-}erase}^I_\ell(!K(u)^{\tau_{co}}) = \mathit{ck\text{-}erase}^I_\ell(!K(u)^{\tau_{co}})$ and hence we get $o \sim_\ell o'$ by the definition of $\sim_\ell$. The claim follows using $(1), (2)$ and reflexivity.

· Otherwise rule (O-Complete) is used on P and P' and

$$\begin{array}{lcl} Q & = & \langle K, N, H, \mathsf{wait}, [\,] \rangle \\ Q' & = & \langle K', N', H', \mathsf{wait}, [\,] \rangle \end{array}$$

and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.

∗ For all other forms of e the rule (O-Complete) is used on P and P' and

$$\begin{aligned} Q &= \langle K, N, H, \mathsf{wait}, [\,] \rangle \\ Q' &= \langle K', N', H', \mathsf{wait}, [\,] \rangle \end{aligned}$$

and $o = o' = \bullet$. The claim follows using $(1), (2), (3)$ and reflexivity.

□

(I-LOAD1)
$$\frac{\Gamma, f \rhd \mathit{transfer}(\mathsf{load}, \mathit{tag}(u), -) \rightsquigarrow (\tau_n, -, \tau_{co}, -) \quad \forall \mathsf{ck}(k,v)^{\tau} \in K(\mathit{host}(u)) : \Gamma, f \rhd \mathit{transfer}(\mathsf{get}, \tau_{co}, \tau) \rightsquigarrow (-,-,-,-)}{\langle K, N, H, \mathsf{wait}, [\,]\rangle \xmapsto{\mathsf{load}(u)}_{\Gamma} \langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, \mathsf{doc\_req}(u : !K(u)^{\tau_{co}})\rangle}$$

(I-LOAD2)
$$\frac{\nexists \tau_n, \tau_{co} : \Gamma, f \rhd \mathit{transfer}(\mathsf{load}, \mathit{tag}(u), -) \rightsquigarrow (\tau_n, -, \tau_{co}, -)}{\langle K, N, H, \mathsf{wait}, [\,]\rangle \xmapsto{\mathsf{load}(u)}_{\Gamma} \langle K, N, H, \mathsf{wait}, \star\rangle}$$

(I-LOAD3)
$$\frac{\Gamma, f \rhd \mathit{transfer}(\mathsf{load}, \mathit{tag}(u), -) \rightsquigarrow (\tau_n, -, \tau_{co}, -) \quad \exists \mathsf{ck}(k,v)^{\tau} \in K(\mathit{host}(u)) : \nexists t : \Gamma, f \rhd \mathit{transfer}(\mathsf{get}, \tau, \tau_{co}) \rightsquigarrow t}{\langle K, N, H, \mathsf{wait}, [\,]\rangle \xmapsto{\mathsf{load}(u)}_{\Gamma} \langle K, N, H, \mathsf{wait}, \star\rangle}$$

(I-DOCRESP1)
$$\frac{\Gamma, f \rhd \mathit{transfer}(\mathsf{doc\_resp}, \tau_n, -) \rightsquigarrow (-, \tau_{ci}, -, \tau_s) \quad \forall \mathsf{ck}(k,v)^{\tau} \in \mathit{CK} : \Gamma, f \rhd \mathit{transfer}(\mathsf{set}, \tau_{co}, \tau) \rightsquigarrow (-,-,-,-)}{\langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, [\,]\rangle \xmapsto{\mathsf{doc\_resp}_n(u:\mathit{CK},e)}_{\Gamma} \langle K \xleftarrow{\mathit{CK}}_{\tau_{ci}} u, N, H, \lceil e \rfloor^{\tau_s}_{@u}, [\,]\rangle}$$

(I-DOCRESP2)
$$\frac{\nexists \tau_{ci}, \tau_s : \Gamma, f \rhd \mathit{transfer}(\mathsf{doc\_resp}, \tau_n, -) \rightsquigarrow (-, \tau_{ci}, -, \tau_s)}{\langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, [\,]\rangle \xmapsto{\mathsf{doc\_resp}_n(u:\mathit{CK},e)}_{\Gamma} \langle K, N, H, \mathsf{wait}, \star\rangle}$$

(I-DOCRESP3)
$$\frac{\Gamma, f \rhd \mathit{transfer}(\mathsf{doc\_resp}, \tau_n, -) \rightsquigarrow (-, \tau_{ci}, -, \tau_s) \quad \exists \mathsf{ck}(k,v)^{\tau} \in \mathit{CK} : \nexists t : \Gamma, f \rhd \mathit{transfer}(\mathsf{set}, \tau_{co}, \tau) \rightsquigarrow t}{\langle K, N \uplus \{n^{\tau_n} : u\}, H, \mathsf{wait}, [\,]\rangle \xmapsto{\mathsf{doc\_resp}_n(u:\mathit{CK},e)}_{\Gamma} \langle K, N, H, \mathsf{wait}, \star\rangle}$$

(O-GETCOOKIE1)
$$\frac{\exists \tau, v : \mathsf{ck}(k,v)^{\tau} \in K(\mathit{host}(u)) \qquad \Gamma, f \rhd \mathit{transfer}(\mathsf{get}, \tau_{co}, \tau) \rightsquigarrow (-,-,-,-)}{\langle K, N, H, \lceil \mathsf{get\text{-}ck}(k) \rfloor^{\tau_s}_{@u}, [\,]\rangle \overset{\bullet}{\mapsto}_{\Gamma} \langle K, N, H, \lceil v \rfloor^{\tau_s}_{@u}, [\,]\rangle}$$

(O-SETCOOKIE1)
$$\frac{\tau = \kappa(\mathit{host}(u), k) \qquad \mathit{CK} = \{\mathsf{ck}(k,v)^{\tau}\} \qquad \Gamma, f \rhd \mathit{transfer}(\mathsf{set}, \tau_s, \tau) \rightsquigarrow (-,-,-,-)}{\langle K, N, H, \lceil \mathsf{set\text{-}ck}(k,v) \rfloor^{\tau_s}_{@u}, [\,]\rangle \overset{\bullet}{\mapsto}_{\Gamma} \langle K \xleftarrow{\mathit{CK}}_{\tau_s} u, N, H, \lceil \mathsf{unit} \rfloor^{\tau_s}_{@u}, [\,]\rangle}$$

(O-GETCOOKIE2)
$$\frac{\exists \tau, v : \mathsf{ck}(k,v)^{\tau} \in K(\mathit{host}(u)) \quad \nexists t : \Gamma, f \rhd \mathit{transfer}(\mathsf{get}, \tau_{co}, \tau) \rightsquigarrow t}{\langle K, N, H, \lceil \mathsf{get\text{-}ck}(k) \rfloor^{\tau_s}_{@u}, [\,]\rangle \overset{\star}{\mapsto}_{\Gamma} \langle K, N, H, \mathsf{wait}, [\,]\rangle}$$

(O-SETCOOKIE2)
$$\frac{\tau = \kappa(\mathit{host}(u), k) \quad \nexists t : \Gamma, f \rhd \mathit{transfer}(\mathsf{set}, \tau_s, \tau) \rightsquigarrow t}{\langle K, N, H, \lceil \mathsf{set\text{-}ck}(k,v) \rfloor^{\tau_s}_{@u}, [\,]\rangle \overset{\star}{\mapsto}_{\Gamma} \langle K, N, H, \mathsf{wait}, [\,]\rangle}$$

Table A.4: Failure semantics of FF^{τ} (excerpt)

APPENDIX B

Appendix to Chapter 5

B.1 Additional Formal Details

B.1.1 Browser Model

The syntax of the scripting language supported in our browser model is given in Table B.1. We let be range over expressions including references (for cookies), values, DOM elements, and binary operations defined over expressions, e.g., arithmetic and logical operations. In particular, expression $\mathsf{dom}(be, be')$ extracts a value from the DOM of the page where the script is running: the expression be identifies the tag of the form in the page, while be' specifies the parameter of interest in the form. For simplicity, we stipulate that $\mathsf{dom}(be, be')$ selects the URL of the form if be' evaluates to 0.

Command **skip** does nothing, while $s; s'$ denotes the standard command concatenation. Command $r := be$ assigns to reference r the value obtained by evaluating the expression be. Command $\mathbf{include}(u, \vec{be})$ retrieves the script located at URL u providing $\vec{be}$ as parameters: we use this construct to model both contents inclusion and a simplified version of XHR requests which is not subject to SOP restrictions which are applied by real browsers. Command $\mathbf{setdom}(be', u, \vec{be})$ substitutes a form in a page, where be' is the tag of the form to be replaced, u and $\vec{be}$ are respectively the URL and the parameters of the new form.

The state of a browser is $(N, M, P, T, Q, \vec{a})^{\iota_b}$ where ι_b is the identity of the user who wants to perform the list of actions $\vec{a}$. The network store N maps connection identifiers to triples (tab, u, l) where tab identifies the tab that initiated the connection, u is the contacted endpoint and l is the origin that has been sent in the `Origin` header of the request and it is needed to correctly handle the header during redirects. M is the cookie jar of the browser, which is modeled as a map from references to values. P maps tab identifiers to pairs $(u, page)$ representing the URL and the contents of the web page and T tracks running scripts: if $T = \{tab \mapsto s\}$, script s is running on the page contained in tab. Finally, Q is a queue (of maximum size 1) of browser requests that is needed to handle redirects in our model.

Browsers			
Expressions	be	::=	$x \mid r \mid v \mid \mathsf{dom}(be, be') \mid be \odot be'$
Scripts	s	::=	$\mathbf{skip} \mid s; s' \mid r := be \mid \mathbf{include}(u, \vec{be}) \mid \mathbf{setdom}(be', u, \vec{be})$
Connections	N	::=	$\{\} \mid \{n \mapsto (tab, u, l)\}$
Pages	P	::=	$\{\} \mid P \uplus \{tab \mapsto (u, page)\}$
Tasks	T	::=	$\{\} \mid \{tab \mapsto s\}$
Output queue	Q	::=	$\{\} \mid \{\alpha\}$
Browsers	B	::=	$(N, M, P, T, Q, \vec{a})^{\iota}$

Table B.1: Syntax of browsers.

Finally, we presuppose the existence of the set of domains $\Delta \subseteq \mathcal{D}$ containing all domains where HSTS is enabled, which essentially models the HSTS preload list[1] that is shipped with modern browsers.

B.1.2 More on Cookie Labels

Now we resume the discussion about the labelling of cookies that we started in subsection 5.3.3.

When a cookie is set with a `Domain` attribute whose value is a domain d, the cookie will be attached to all requests towards d and its subdomains. This behavior is modelled by the labelling

$$(\bigwedge_{d' \leq d} \mathsf{http}(d') \wedge \mathsf{https}(d'), \bigwedge_{d' \sim d} (\mathsf{http}(d') \wedge \mathsf{https}(d')))$$

where $\leq$ is a preorder defined on $\mathcal{D}$ such that $d \leq d'$ iff d is subdomain of d'.

We discuss now the impact of HSTS on cookie labels: since this security policy prevents browsers from communicating with certain domains over HTTP, essentially it prevents network attackers from setting cookies by modifying HTTP responses coming from these domains. In particular, the label for a `Secure` cookie for domain d becomes the following:

$$(\mathsf{https}(d), \bigwedge_{\substack{d' \sim d \\ d' \notin \Delta}} \mathsf{http}(d') \wedge \bigwedge_{d' \sim d} \mathsf{https}(d')))$$

If HSTS is enabled for d and all its related domains, then the cookie label is the same as that of cookies with the `__Secure-` cookie prefix, i.e.:

$$(\mathsf{https}(d), \bigwedge_{d' \sim d} \mathsf{https}(d'))$$

B.1.3 Complete Semantics

Browsers

We present the browser semantics in Table B.2 where we exclude non-deterministic behaviors by requiring that *i)* at most one network connection is open at any time; *ii)* the user performs

[1] `https://hstspreload.org`

Expressions

(BE-VAL)

$$\frac{}{eval_\ell(v, M, f) = v}$$

(BE-BINOP)

$$\frac{eval_\ell(be, M, f) = v \qquad eval_\ell(be', M, f) = v'}{eval_\ell(be \odot be', M, f) = v \odot v'}$$

(BE-REFERENCE)

$$\frac{C(\lambda(r)) \sqsubseteq_C C(\ell)}{eval_\ell(r, M, f) = M(r)}$$

(BE-DOM)

$$\frac{\begin{array}{c} eval_\ell(be, M, f) = v' \qquad eval_\ell(be', M, f) = v'' \\ \{v' \mapsto \mathsf{form}(u, \vec{v})\} \in f \qquad v'' = 0 \Rightarrow v''' = u \qquad v'' \neq 0 \Rightarrow v''' = v_{v''} \end{array}}{eval_\ell(\mathsf{dom}(be, be'), M, f) = v'''}$$

Browser

(B-LOAD)

$$\frac{\begin{array}{c} n \leftarrow \mathcal{N} \\ ck = get\_ck(M, u) \qquad \alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, \bot) \qquad (\mathsf{orig}(u) = \mathsf{http}(d) \Rightarrow d \notin \Delta) \end{array}}{(\{\}, M, P, \{\}, \{\}, \mathsf{load}(tab, u, p) :: \vec{a})^{\iota_b} \xrightarrow{\bullet} (\{n \mapsto (tab, u, \bot)\}, M, P, \{\}, \{\alpha\}, \vec{a})^{\iota_b}}$$

(B-INCLUDE)

$$\frac{\begin{array}{c} n \leftarrow \mathcal{N} \qquad ck = get\_ck(M, u) \qquad \{tab \mapsto (u', f)\} \in P \\ \forall k \in [1 \ldots |\vec{be}|] : p(k) = eval_{\lambda(u')}(be_k, M, f) \\ \alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, \mathsf{orig}(u')) \qquad (\mathsf{orig}(u') = \mathsf{http}(d) \Rightarrow d \notin \Delta) \end{array}}{\begin{array}{c} (\{\}, M, P, \{tab \mapsto \mathbf{include}(u, \vec{be})\}, \{\}, \vec{a})^{\iota_b} \xrightarrow{\bullet} \\ (\{n \mapsto (tab, u, \mathsf{orig}(u'))\}, M, P, \{tab \mapsto \mathbf{skip}\}, \{\alpha\}, \vec{a})^{\iota_b} \end{array}}$$

(B-RECVLOAD)

$$\frac{\begin{array}{c} \alpha = \mathsf{res}(n, u, \bot, \_, ck, page, s) \\ M' = upd\_ck(M, u, ck) \qquad \vec{a}\,' = (page = \mathsf{error})\ ?\ (\mathsf{halt} :: \vec{a}) : \vec{a} \end{array}}{(\{n \mapsto (tab, u, o)\}, M, P, \{\}, \{\}, \vec{a})^{\iota_b} \xrightarrow{\alpha} (\{\}, M', P \triangleleft \{tab \mapsto (u, page)\}, \{tab \mapsto s\}, \{\}, \vec{a}\,')^{\iota_b}}$$

(B-RECVINCLUDE)

$$\frac{\alpha = \mathsf{res}(n, u, \bot, \_, ck, page, s) \qquad M' = upd\_ck(M, u, ck)}{(\{n \mapsto (tab, u, o)\}, M, P, \{tab \mapsto s'\}, \{\}, \vec{a})^{\iota_b} \xrightarrow{\alpha} (\{\}, M', P, \{tab \mapsto s; s'\}, \{\}, \vec{a})^{\iota_b}}$$

Table B.2: Semantics of browsers.

Browser (continued)

(B-REDIRECT)

$$\frac{\alpha = \mathsf{res}(n,u,u',\vec{v},ck,\_,\_) \qquad M' = upd\_ck(M,u,ck) \qquad n' \leftarrow \mathcal{N} \qquad ck' = get\_ck(M',u') \qquad \forall k \in [1\ldots|\vec{v}|] : p(k) = v_k \qquad o' = (o = \mathrm{orig}(u))\ ?\ o : \bot \qquad \alpha' = \overline{\mathsf{req}}(\iota_b,n',u',p,ck',o') \qquad (\mathrm{orig}(u') = \mathsf{http}(d) \Rightarrow d \notin \Delta)}{(\{n \mapsto (tab,u,o)\},M,P,\{\},\{\},\vec{a})^{\iota_b} \xrightarrow{\alpha} (\{n' \mapsto (tab,u',o')\},M',P,\{\},\{\alpha'\},\vec{a})^{\iota_b}}$$

(B-SUBMIT)

$$\frac{\{tab \mapsto (u,f)\} \in P \qquad \{v' \mapsto \mathsf{form}(u',\vec{v})\} \in f \qquad \forall k \in [1\ldots|\vec{v}|].\, p'(k) = k \in dom(p)\ ?\ p(k) : v_k \qquad n \leftarrow \mathcal{N} \qquad ck = get\_ck(M,u') \qquad \alpha = \overline{\mathsf{req}}(\iota_b,n,u',p',ck,\mathrm{orig}(u)) \qquad (\mathrm{orig}(u') = \mathsf{http}(d) \Rightarrow d \notin \Delta)}{(\{\},M,P,\{\},\{\},\mathsf{submit}(tab,u,v',p) :: \vec{a})^{\iota_b} \xrightarrow{\bullet} (\{n \mapsto (tab,u',\mathrm{orig}(u))\},M,P,\{\},\{\alpha\},\vec{a})^{\iota_b}}$$

(B-FLUSH)

$$\frac{}{(N,M,P,T,\{\alpha\},\vec{a})^{\iota_b} \xrightarrow{\alpha} (N,M,P,T,\{\},\vec{a})^{\iota_b}}$$

(B-SEQ)

$$\frac{(\{\},M,P,\{tab \mapsto s\},\{\},\vec{a})^{\iota_b} \xrightarrow{\alpha} (\{\},M',P',\{tab \mapsto s'\},\{\},\vec{a})^{\iota_b}}{(\{\},M,P,\{tab \mapsto s;s''\},\{\},\vec{a})^{\iota_b} \xrightarrow{\alpha} (\{\},M',P',\{tab \mapsto s';s''\},\{\},\vec{a})^{\iota_b}}$$

(B-SKIP)

$$\frac{}{(\{\},M,P,\{tab \mapsto \mathbf{skip};s\},\{\},\vec{a})^{\iota_b} \xrightarrow{\bullet} (\{\},M,P,\{tab \mapsto s\},\{\},\vec{a})^{\iota_b}}$$

(B-END)

$$\frac{}{(\{\},M,P,\{tab \mapsto \mathbf{skip}\},\{\},\vec{a})^{\iota_b} \xrightarrow{\bullet} (\{\},M,P,\{\},\{\},\vec{a})^{\iota_b}}$$

(B-SETREFERENCE)

$$\frac{\{tab \mapsto (u,f)\} \in T \qquad \ell = \lambda(u) \qquad eval_\ell(be,M,f) = v \qquad I(\ell) \sqsubseteq_I I(\lambda(r))}{(\{\},M,P,\{tab \mapsto r := be\},\{\},\vec{a})^{\iota_b} \xrightarrow{\bullet} (\{\},M\{r \mapsto v\},P,\{tab \mapsto \mathbf{skip}\},\{\},\vec{a})^{\iota_b}}$$

(B-SETDOM)

$$\frac{\ell = \lambda(u') \qquad eval_\ell(be',M,f) = v' \qquad \forall k \in [1\ldots|\vec{be}|].\, v_k = eval_\ell(be_k,M,f)}{(\{\},M,P \uplus \{tab \mapsto (u',f)\},\{tab \mapsto \mathbf{setdom}(be',u,\vec{be})\},\{\},\vec{a})^{\iota_b} \xrightarrow{\bullet} (\{\},M,P \uplus \{tab \mapsto (u',f\{v' \mapsto \mathsf{form}(u,\vec{v})\})\},\{tab \mapsto \mathbf{skip}\},\{\},\vec{a})^{\iota_b}}$$

Table B.2: Semantics of browsers (continued).

an action only when there are no pending network connections and no script is running, which amounts to asking that the user waits that the current page is completely rendered. This design choice is made to simplify our security proof and it has no impact the expressiveness of our model.

First we define the semantics of expressions in terms of the function $eval_\ell(be, M, f)$ that evaluates the expression be in terms of the cookie jar M, the DOM of the webpage f and the security context ℓ. Rule (BE-REFERENCE) models the access to the cookie jar, which is allowed only if the confidentiality level of the reference is below that of the security context. Rule (BE-DOM) selects a value from the DOM of the page depending on the values of the expressions be and be'. Rules (BE-VAL) and (BE-BINOP) are standard.

Our semantics relies on the auxiliary functions get_ck and upd_ck to select the cookies to be attached to an outgoing request and to update the cookie jar with the cookies provided in an incoming response, respectively. Given a cookie jar M and a URL u, we let $get\_ck(M, u)$ be the map ck such that $ck(r) = v$ iff $M(r) = v$ and $C(\lambda(r)) \sqsubseteq_C C(\lambda(u))$. Given a cookie jar M, a URL u and a map of cookies ck, we let $upd\_ck(M, u, ck) = M \triangleleft (ck \uparrow u)$ where $ck \uparrow u$ is the map ck' such that $ck'(r) = v$ iff $ck(r) = v$ and $I(\lambda(u)) \sqsubseteq_I I(\lambda(r))$.

We describe now the rules of the browser semantics. Rule (B-LOAD) models the loading of a new page as dictated by the action $\mathsf{load}(tab, u, p)$. The browser opens a new network connection represented by the fresh name n and sends a request to the server located at u providing the parameters p and attaching the cookies ck selected from the cookie jar, with an empty origin header, as represented by the action $\overline{\mathsf{req}}(\iota_b, n, u, p, ck, \bot)$. If the protocol of the URL u is HTTP, we only allow the request if HSTS is not activated for the domain. In the connections store we associate n to the triple $(tab, u, \bot)$. Similarly, rule (B-INCLUDE) models the embedding of a script with the **include** directive of our scripting language. Compared to (B-LOAD), the main differences are that *i)* the list of expressions $\vec{be}$ specified in the instruction are evaluated; *ii)* the request contains the origin of the page where the script is executed. Notice that the execution of the script is paused until a response is received: this behavior is similar to what happens in standard browsers when embedding scripts or using synchronous XHR requests.

Rule (B-RECVLOAD) models the receiving of a webpage over a pending network connection, represented by the transition label $\mathsf{res}(n, u, \bot, \_, ck, page, s)$. As a result, the connection n is closed, the cookie jar is updated with the cookies ck attached to the response, the content of the tab associated to n is replaced with the received page and the script s is executed in that tab. In case the page error is received, we prepend the action halt to the list of user actions: since this action is not be consumed by any of the semantic rules, this models a cautious user that interrupts the navigation when an unexpected error occurs during the navigation. Rule (B-RECVINCLUDE) is similar to the previous rule: the main differences are that *i)* the page contained in tab is left unchanged and the one sent by the server is discarded, therefore the user continues interacting with the website even when the error page is received by the browser; *ii)* the script s sent by the server is prepended to the script s' that is waiting to run on the page. Rule (B-REDIRECT) models the receiving of a redirect from the server to URL u' with parameters $\vec{v}$, represented by the transition label $\mathsf{res}(n, u, u', \vec{v}, ck, \_, \_)$. The cookie jar is updated with the cookies ck set in the response and a new request to u' with the appropriate cookies and parameters is prepared

by the browser and added to the output queue. If the origin o of the original request matches the origin $\text{orig}(u')$ of the new target, the origin header remains the same for the new request, otherwise it is set to $\perp$. The redirect is only allowed if it respects the HSTS settings for the new target.

Rule (B-SUBMIT) models the user clicking on a link or submitting a form in the page identified by URL u which is currently open in the browser at the specified tab. For each parameter we first check if the user has inserted a value by inspecting the map p, otherwise we fallback to the pre-filled parameter contained in the form. A new network connection is opened, cookies from the cookie jar are attached to the outgoing request and the HSTS settings are checked as in (B-LOAD). The origin of the request is the origin of the URL u of the page containing the form. Rule (B-FLUSH) outputs on the network the request in the output queue produced by rules (B-LOAD), (B-INCLUDE) (B-REDIRECT) and (B-SUBMIT).

The remaining rules describe how scripts are processed. Rule (B-SEQ) models sequencing of script commands, (B-SKIP) processes the **skip** command and (B-END) terminates the script execution. Rule (B-SETREFERENCE) models the setting of a cookie by a script, which is allowed if the integrity label of the reference is above that of the URL of the page where the script is running. Finally, rule (B-SETDOM) models the update of a form in the DOM of the page where the script is running.

Servers

In Table B.3 we give the rules of the server semantics that were not presented in subsection 5.3.4. Rule (S-SEQ) is used for sequencing commands, (S-SKIP) to evaluate **skip**, (S-IFTRUE) and (S-IFFALSE) for conditionals, (S-OCHKFAIL) and (S-TCHKSUCC) cover the missing cases of origin and token check, (S-SETGLOBAL) and (S-SETSESSION) respectively update the value of a reference in the global memory and in the session memory. Rule (S-REDIRECT) models a redirect from the server to the URL u' with parameters $\vec{z}$ that sets the cookies ck in the user's browser. The page and script components of the action $\overline{\text{res}}$ are respectively the empty page and the empty script, as they will be anyway discarded by the browser. As in rule (S-REPLY) shown in Table 5.2, all occurrences of variables in $\vec{x}$ contained in the response are replaced with the results of the evaluations of the corresponding expressions in $\vec{se}$ and we stipulate that the execution terminates after sending the message. Finally, rules (S-LPARALLEL) and (S-RPARALLEL) handle the parallel composition of threads.

Web Systems

We report in Table B.4 the rules of the web systems semantics that were not presented in the body of the paper. Rules (W-LPARALLEL) and (W-RPARALLEL) model the parallel composition of web systems. Rule (A-NIL) is applied when no synchronizations between two entities occur.

Rule (A-SERBRO) models an honest server providing a response to a browser over a pending connection. Here the knowledge of the attacker is extended either if she can read the messages using her network capabilities. Rule (A-SERATK) models the reception of a response from an honest server by the attacker. We require that the attacker knows the connection identifier n to

Expressions

$$\text{(SE-VAL)}\ \frac{}{eval_E(v, D) = v} \qquad \text{(SE-BINOP)}\ \frac{eval_E(se, D) = v \qquad eval_E(se', D) = v'}{eval_E(se \odot se', D) = v \odot v'}$$

$$\text{(SE-READGLOBAL)}\ \frac{}{eval_{i,\_}(@r, D) = D(i, r)} \qquad \text{(SE-READSESSION)}\ \frac{}{eval_{i,j}(\$r, D) = D(j, r)} \qquad \text{(SE-FRESH)}\ \frac{n \leftarrow \mathcal{N}}{eval_E(\mathit{fresh}(), D) = n}$$

Table B.3: Semantics of servers (remaining rules).

prevent her from intercepting arbitrary traffic and we extend her knowledge with the contents of the message. Rule (A-ATKBRO) models the attacker providing a response to a browser either using her network capabilities or a server under her control. In this case we require that the attacker is able to produce the contents of the response using her knowledge $\mathcal{K}$, which amounts to asking that all names in the response are known to the attacker.

Finally, rule (A-TIMEOUT) is used to process requests to endpoints not present in the system W (e.g., attacker-controlled endpoints in a run without the attacker): in such a case, we let the browser process an empty response.

B.1.4 Typing Rules for Scripts

Table B.5 presents the typing rules that were not introduced in the body of the paper due to lack of space.

Browser Expressions

Typing of browser expressions is ruled by the judgement $\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} be : \tau$, meaning that the expression se has type τ in the typing environment Γ and typing branch b. Rules are similar to those for server expressions, but in this case we do not carry around the session label since there are no session references. Rule (T-BEDOM) is used to type reading data from the DOM, where we conservatively forbid reading from the DOM in the honest branch and use label ℓ_a otherwise, since we then know that the type of all values in the DOM is upper bounded by ℓ_a.

Browser References

Typing of references in the browser is ruled by the judgment $\Gamma \vdash^{\mathsf{br}}_{\ell_a} r : \mathtt{ref}(\tau)$ meaning that the reference r has reference type $\mathtt{ref}(\tau)$ in the environment Γ. Compared to server references, the main difference is that there are no session references on the browser side.

Server

(S-SEQ)
$$\frac{(D,\phi,\lceil c\rfloor^R_E) \xrightarrow{\alpha} (D',\phi',\lceil c'\rfloor^R_{E'})}{(D,\phi,\lceil c;c''\rfloor^R_E) \xrightarrow{\alpha} (D',\phi',\lceil c';c''\rfloor^R_{E'})}$$

(S-SKIP)
$$\frac{}{(D,\phi,\lceil \textbf{skip};c\rfloor^R_E) \xrightarrow{\bullet} (D,\phi,\lceil c\rfloor^R_E)}$$

(S-IFTRUE)
$$\frac{eval_E(se,D) = true}{(D,\phi,\lceil \textbf{if } se \textbf{ then } c \textbf{ else } c'\rfloor^R_E) \xrightarrow{\bullet} (D,\phi,\lceil c\rfloor^R_E)}$$

(S-IFFALSE)
$$\frac{eval_E(se,D) = false}{(D,\phi,\lceil \textbf{if } se \textbf{ then } c \textbf{ else } c'\rfloor^R_E) \xrightarrow{\bullet} (D,\phi,\lceil c'\rfloor^R_E)}$$

(S-OCHKFAIL)
$$\frac{R = n,u,\iota_b,o \qquad o \notin O}{(D,\phi,\lceil \textbf{if originchk}(O) \textbf{ then } c\rfloor^R_E) \xrightarrow{\mathsf{error}} (D,\phi,\lceil \textbf{reply } (\mathsf{error},\textbf{skip},\{\})\rfloor^R_E)}$$

(S-TCHKSUCC)
$$\frac{eval_E(e_1,D) = eval_E(e_2,D)}{(D,\phi,\lceil \textbf{if tokenchk}(e_1,e_2) \textbf{ then } c\rfloor^R_E) \xrightarrow{\bullet} (D,\phi,\lceil c\rfloor^R_E)}$$

(S-SETGLOBAL)
$$\frac{E = i,\_ \qquad eval_E(se,D) = v}{(D,\phi,\lceil @r := se\rfloor^R_E) \xrightarrow{\bullet} (D\{i \mapsto D(i)\{r \mapsto v\}\},\phi,\lceil \textbf{skip}\rfloor^R_E)}$$

(S-SETSESSION)
$$\frac{eval_{i,j}(se,D) = v}{(D,\phi,\lceil \$r := se\rfloor^R_{i,j}) \xrightarrow{\bullet} (D\{j \mapsto D(j)\{r \mapsto v\}\},\phi,\lceil \textbf{skip}\rfloor^R_{i,j})}$$

(S-REDIRECT)
$$\frac{\begin{array}{c}R = n,u,\iota_b,l \qquad m = |\vec{x}| = |\vec{se}| \qquad \forall k \in [1,m].\ eval_E(se_k,D) = v_k \\ \sigma = [x_1 \mapsto v_1,\ldots,x_m \mapsto v_m] \qquad \alpha = \overline{\mathsf{res}}(n,u,u',\vec{z}\sigma,ck\sigma,\{\},\textbf{skip})\end{array}}{(D,\phi,\lceil \textbf{redirect } (u',\vec{z},ck) \textbf{ with } \vec{x} = \vec{se}\rfloor^R_E) \xrightarrow{\alpha} (D,\phi,\lceil \textbf{halt}\rfloor^R_E)}$$

(S-LPARALLEL)
$$\frac{(D,\phi,t) \xrightarrow{\alpha} (D',\phi',t'')}{(D,\phi,t \parallel t') \xrightarrow{\alpha} (D',\phi',t'' \parallel t')}$$

(S-RPARALLEL)
$$\frac{(D,\phi,t') \xrightarrow{\alpha} (D',\phi',t'')}{(D,\phi,t \parallel t') \xrightarrow{\alpha} (D',\phi',t \parallel t'')}$$

Table B.3: Semantics of servers (remaining rules, continued).

(W-LPARALLEL)
$$\frac{W \xrightarrow{\alpha} W'}{W \parallel W'' \xrightarrow{\alpha} W' \parallel W''}$$

(W-RPARALLEL)
$$\frac{W \xrightarrow{\alpha} W'}{W'' \parallel W \xrightarrow{\alpha} W'' \parallel W'}$$

(A-NIL)
$$\frac{W \xrightarrow{\alpha} W' \qquad \alpha \in \{\bullet, \sharp[\vec{v}]_{\ell'}^{\iota_b,\iota_s}\}}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell,\mathcal{K}) \triangleright W'}$$

(A-SERBRO)
$$\frac{W \xrightarrow{\mathsf{res}(n,u,u',\vec{v},ck,page,s)} W' \qquad W' \xrightarrow{\overline{\mathsf{res}}(n,u,u',\vec{v},ck,page,s)} W'' \qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell)) \; ? \; (\mathcal{K} \cup ns(ck,page,s,\vec{v})) : \mathcal{K}}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\bullet} (\ell,\mathcal{K}') \triangleright W''}$$

(A-SERATK)
$$\frac{n \in \mathcal{K} \qquad \alpha = \overline{\mathsf{res}}(n,u,u',\vec{v},ck,page,s) \qquad W \xrightarrow{\alpha} W' \qquad \mathcal{K}' = \mathcal{K} \cup ns(ck,page,s,\vec{v})}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell,\mathcal{K}') \triangleright W'}$$

(A-ATKBRO)
$$\frac{\alpha = \mathsf{res}(n,u,u',\vec{v},ck,page,s) \qquad W \xrightarrow{\alpha} W' \qquad I(\ell) \sqsubseteq_I I(\lambda(u)) \qquad \{n\} \cup ns(ck,page,s,\vec{v}) \subseteq \mathcal{K}}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell,\mathcal{K}) \triangleright W'}$$

(A-TIMEOUT)
$$\frac{W \xrightarrow{\overline{\mathsf{req}}(\iota_b,n,u,p,ck,o)} W' \qquad W' \xrightarrow{\mathsf{req}(\iota_b,n,y,p,ck,o)} \!\!\!\!\!\!\diagup \qquad W' \xrightarrow{\mathsf{res}(n,u,\bot,\{\},\{\},\{\},\mathbf{skip})} W'' \qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell)) \; ? \; (\mathcal{K} \cup ns(p,ck)) : \mathcal{K}}{(\ell,\mathcal{K}) \triangleright W \xrightarrow{\bullet} (\ell,\mathcal{K}') \triangleright W''}$$

Table B.4: Semantics of web systems (remaining rules).

Scripts

The typing judgment for scripts $\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a,\mathcal{P}} s$ reads as follows: the script s is well-typed in the environment Γ under the program counter label $\mathtt{pc}$ in the typing branch b.

Three straight-forward to type scripts are (T-BSKIP) that trivially does nothing, (T-BSEQ) checks both the concatenated commands and (T-BASSIGN) handles reference assignments just like (T-SETGLOBAL).

In the honest branch, (T-BSETDOM) performs the same checks as (T-FORM), namely that the script with program counter label $\mathtt{pc}$ is allowed to trigger a request to URL u, that the parameters of the generated form respect the type of the URL, and that the type associated to the name of the form matches the type of the URL. For the attacked case, we just require that all parameters have type ℓ_a, as in the CSRF branch in rule (T-REPLY). Notice that we restrict the first expression in **setdom** to be a value, so that we can statically look up the associated type in $\Gamma_{\mathcal{V}}$.

Browser expressions and references

$$\text{(T-BEVAR)}\ \frac{}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} x : \Gamma_{\mathcal{X}}(x)} \qquad \text{(T-BEREF)}\ \frac{}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} r : \Gamma_{\mathcal{R}^@}(r)} \qquad \text{(T-BEVAL)}\ \frac{v \notin \mathcal{N}}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} v : \bot} \qquad \text{(T-BEUNDEF)}\ \frac{}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} \bot : \tau}$$

$$\text{(T-BENAME)}\ \frac{}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} n^{\ell} : \mathtt{cred}(\ell)} \qquad \text{(T-BEDOM)}\ \frac{b \neq \mathsf{hon}}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} \mathsf{dom}(be, be') : \ell_a}$$

$$\text{(T-BEBINOP)}\ \frac{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} se : \tau \qquad \Gamma, b \vdash^{\mathsf{be}}_{\ell_a} se' : \tau' \qquad (\tau = \ell \wedge \tau = \ell') \vee \odot \text{ is } =}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} se \odot se' : \mathit{label}(\tau) \sqcup \mathit{label}(\tau')} \qquad \text{(T-BESUB)}\ \frac{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} be : \tau' \qquad \tau' \sqsubseteq_{\ell_a} \tau}{\Gamma, b \vdash^{\mathsf{be}}_{\ell_a} be : \tau}$$

$$\text{(T-BREF)}\ \frac{}{\Gamma \vdash^{\mathsf{br}}_{\ell_a} r : \Gamma_{\mathcal{R}^@}(r)} \qquad \text{(T-BRSUB)}\ \frac{\Gamma \vdash^{\mathsf{br}}_{\ell_a} r : \mathtt{ref}(\tau') \qquad \tau \sqsubseteq_{\ell_a} \tau'}{\Gamma \vdash^{\mathsf{br}}_{\ell_a} r : \mathtt{ref}(\tau)}$$

Scripts

$$\text{(T-BSEQ)}\ \frac{\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} s \qquad \Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} s'}{\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} s; s'} \qquad \text{(T-BSKIP)}\ \frac{}{\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} \mathbf{skip}}$$

$$\text{(T-BASSIGN)}\ \frac{\Gamma \vdash^{\mathsf{br}}_{\ell_a} r : \mathtt{ref}(\tau) \qquad \Gamma, b \vdash^{\mathsf{be}}_{\ell_a} be : \tau \qquad \mathtt{pc} \sqsubseteq_I I(\tau)}{\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} r := be}$$

$$\text{(T-BSETDOM)}\ \frac{\begin{array}{c}\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad m = |\vec{be}| = |\vec{\tau}| \qquad \forall k \in [1 \ldots m].\, \Gamma, b \vdash^{\mathsf{be}}_{\ell_a} be_k : \tau'_k \\ (b = \mathsf{hon} \Rightarrow \Gamma_{\mathcal{V}}(v) = \Gamma_{\mathcal{U}}(u) \wedge \mathtt{pc} \sqsubseteq_I I(\ell_u) \wedge \forall k \in [1 \ldots m].\, \tau'_k \sqsubseteq_{\ell_a} \tau_k) \\ (b \neq \mathsf{hon} \Rightarrow \forall k \in [1 \ldots m].\, \tau'_k \sqsubseteq_{\ell_a} \ell_a)\end{array}}{\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} \mathbf{setdom}(v, u, \vec{be})}$$

$$\text{(T-BINCLUDE)}\ \frac{\begin{array}{c}\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad m = |\vec{be}| = |\vec{\tau}| \qquad \forall k \in [1 \ldots m].\, \Gamma, b \vdash^{\mathsf{be}}_{\ell_a} be_k : \tau'_k \\ (b = \mathsf{hon} \Rightarrow I(\ell_a) \not\sqsubseteq_I I(\ell_u) \wedge l_r = \mathtt{pc} \wedge \mathtt{pc} \sqsubseteq_I I(\ell_u) \wedge \forall k \in [1 \ldots m].\, \tau'_k \sqsubseteq_{\ell_a} \tau_k \wedge u \notin \mathcal{P}) \\ (b \neq \mathsf{hon} \Rightarrow \forall k \in [1 \ldots m].\, \tau'_k \sqsubseteq_{\ell_a} \ell_a)\end{array}}{\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a, \mathcal{P}} \mathbf{include}(u, \vec{be})}$$

Table B.5: Typing rules for scripts.

Rule (T-BINCLUDE) performs the same checks on the URL parameters as the previous rule, but additionally requires in the honest case that the integrity of the network connection is high to prevent an attacker from injecting her own script which would then be executed in the context of the original page. Furthermore, we require that the included URL is not protected by an origin check as otherwise an attacker could abuse this to indirectly trigger a CSRF with the expected origin. We also require that the expected integrity label of the reply of the included URL u is the same as the $\mathtt{pc}$ used to type the current script: this is needed since executing a script that was typed with a program counter label of higher integrity leads to a privilege escalation, e.g., it could write to a high integrity reference which the current script should not be allowed to do. Including a script of lower integrity is also problematic since we type scripts in the same context as the DOM of the page, thus we would allow a low integrity script to write into the current (high integrity) DOM.

B.1.5 Formal Results

Definition 6. *Let $\vec{a}$ be a list of user actions containing $a_k = \mathsf{load}(tab, u, p)$. The* navigation flow *initiated from a_k is the list of actions $a_k :: nf(\vec{a} \Downarrow k, tab)$ where $\vec{a} \Downarrow k$ is the list obtained from $\vec{a}$ by dropping the first k elements and function nf is defined by the following rules:*

$$nf(\langle\rangle, tab) = \langle\rangle \qquad nf(\mathsf{load}(tab, u, p) :: \vec{a}, tab) = \langle\rangle$$

$$\frac{a = \mathsf{submit}(tab, u, v, p) \qquad nf(\vec{a}, tab) = \vec{a}'}{nf(a :: \vec{a}, tab) = a :: \vec{a}'}$$

$$\frac{a \neq \mathsf{submit}(tab, u, v, p) \qquad a \neq \mathsf{load}(tab, u, p) \qquad nf(\vec{a}, tab) = \vec{a}'}{nf(a :: \vec{a}, tab) = \vec{a}'}$$

B.1.6 Case Studies

Besides the case study on HotCRP that we have presented in the body of the paper, we have also analyzed other two popular PHP applications: phpMyAdmin [php], a software for database administration, and Moodle [Moo], an e-learning platform. We discuss now the encoding of the session management logic in these applications and some session integrity vulnerabilities affecting them, either novel or taken from recent CVEs.

Moodle

we present now the *login* endpoint implementing the authentication logic on Moodle. The endpoint expects the cookie *sid* which is used to store session data and the credentials of the user,

namely the username uid and the password pwd. Its encoding in our calculus is the following:

```
1. login[sid](uid, pwd) ↪
2.   if @sid = ⊥ then
3.     @sid = fresh();
4.   start @sid;
5.   if $uid ≠ ⊥ then
6.     redirect (profile, ⟨⟩, {});
7.   else if uid = ⊥ then
8.     reply ({auth ↦ form(login, ⟨⊥, ⊥⟩)}, skip, {sid ↦ x})
9.       with x = @sid;
10.  else
11.      @sid = fresh(); login uid, pwd, @sid; start @sid;
12.      $uid = uid; $sesskey = fresh();
13.      redirect (profile, ⟨⟩, {sid ↦ x}) with x = @sid;
```

If no cookie sid has been provided, e.g., when the user visits the website for the first time, a fresh cookie is generated (lines 2–3). The session identified by sid is then started (line 4): if the identifier denotes a valid session, session variables stored when processing previous requests are restored. If the user previously authenticated on the website, the session variable $\$uid$ is different from the undefined value $\bot$ and a redirect to the $profile$ endpoint (that here we do not model) is sent to the browser (lines 5–6). If the user is not authenticated and did not provide a pair of credentials, the server replies with a page containing the login form and a new cookie sid is set into the user's browser (lines 7–9). Finally, if the user has provided valid credentials, the endpoint starts a fresh session (to prevent fixation), stores in the session memory the user's identity and a fresh value in $\$sesskey$ which is used to implement CSRF protection, then redirects the user to the $profile$ endpoint and sets the new session identifier in the cookie sid in the user's browser (lines 10–13).

Since $login$ does not perform any origin or token check before performing the **login** command, the endpoint is vulnerable to Login CSRF attacks, as it was the case for Moodle until November 2018 [MIT18b]. As discussed in subsection 5.2.4 for HotCRP, this problem is captured when typing since the cookie must be of low integrity since no CSRF check is performed when it is set, therefore it cannot be used to perform authenticated actions of high integrity.

The solution implemented by Moodle developers uses pre-sessions, as we proposed for HotCRP in section 5.2. In particular, developers decided for convenience to use the same cookie to handle both pre-sessions and sessions: this promotion of the cookie from low integrity, to handle the pre-session, to high integrity, when the session identifier is refreshed after authentication, cannot be modeled in our type system since we have a single static type environment for references, therefore type-checking would fail. In our encoding we model the fix by using two different cookies, pre and sid, which are set to the same value and respectively used in the pre-session and the session. The problem can also be solved in the type system by distinguishing two different

typing environments, but we leave this for future work.

```
1. login[pre](uid, pwd, ltoken) ↪
2.   if @pre = ⊥ then
3.     @pre = fresh();
4.   start @pre;
5.   if $uid ≠ ⊥ then
6.     redirect (profile, ⟨⟩, {});
7.   else if uid = ⊥ then
8.     if $ltoken = ⊥ then
9.       $ltoken = fresh();
10.      reply ({auth ↦ form(login, ⟨⊥, ⊥, x⟩)}, skip, {pre ↦ y})
11.        with x = $ltoken, y = @pre;
12.    else
13.      @ltoken = $ltoken; $ltoken = fresh();
14.      if tokenchk(ltoken, @ltoken) then
15.        @sid = fresh(); login uid, pwd, @sid; start @sid;
16.        $uid = uid; $sesskey = fresh();
17.        redirect (profile, ⟨⟩, {sid ↦ x, pre ↦ y})
18.          with x = @sid, y = @sid;
```

The main differences compared to the previous encoding are the following: *i)* the endpoint now expects a third $ltoken$ which is used to implement CSRF protection (line 1); *ii)* the login form is enriched with a CSRF token which is stored in the pre-session memory (lines 8–11); *iii)* the token stored in the session memory is compared to the one provided by the user before performing the authentication (line 14). After applying the fix, it is possible to perform high integrity authenticated actions within session started from the cookie sid since it is possible to assign it a high integrity credential type when type-checking against the web attacker.

phpMyAdmin

we show now the encoding of the session management logic for phpMyAdmin. In the following we model two HTTPS endpoints hosted on domain d_P: *login*, where database administrators can authenticate using their access credentials, and *drop*, where administrators can remove databases from the system.

We briefly discuss some implementation details of phpMyAdmin before presenting our encoding of the endpoints:

- for CSRF and login CSRF protection, phpMyAdmin inspects all incoming POST requests to check whether they contain a parameter $token$ which is equal to the value stored in the (pre-)session memory;
- the parameters provided by the user are retrieved using the `$_REQUEST` array which allows to uniformly access POST and GET parameters: in our encodings we model this

behavior by using two different variables for each input of interest, e.g., g_pwd and p_pwd for the password when provided via GET or POST, respectively;

- a single cookie is used for pre-sessions and sessions while, as in the case of Moodle, we use two cookies pre and sid;
- upon authentication, username and password are stored encrypted in two cookies: in our model we store them in the clear and use strong cookie labels to provide cookies with the confidentiality and integrity guarantees given by encryption.

We start with the encoding of the *login* endpoint. As parameters it expects the username and the password, both provided via GET and POST, and the login CSRF token, while as cookies we have pre for the pre-session, uid and pwd where the credentials are stored upon authentication. The encoding in our calculus is the following:

```
1. login[pre, uid, pwd](g_uid, p_uid, g_pwd, p_pwd, token) ↪
2.   if @uid ≠ ⊥ and @pwd ≠ ⊥ then
3.     redirect (index, ⟨⟩, {});
4.   if @pre = ⊥ then
5.     @pre = fresh();
6.   start @pre;
7.   if g_uid = ⊥ and p_uid = ⊥ then
8.     $token = fresh();
9.     reply ({auth ↦ form(login, ⟨⊥, ⊥, ⊥, ⊥, x⟩)}, skip,
10.      {pre ↦ y}) with x = $token, y = @pre;
11.  else if p_uid ≠ ⊥ then
12.    if tokenchk(token, $token) then
13.      @sid = fresh(); login p_uid, p_pwd, @sid;
14.      start @sid; $token = fresh();
15.      redirect (index, ⟨⟩, {uid ↦ x, pwd ↦ y, pre ↦ z,
16.        sid ↦ z}) with x = p_uid, y = p_pwd, z = @sid;
17.  else
18.    @sid = fresh(); login uid, pwd, @sid;
19.    start @sid; $token = fresh();
20.    redirect (index, ⟨⟩, {uid ↦ x, pwd ↦ y, pre ↦ z, sid ↦ z})
21.      with x = g_uid, y = g_pwd, z = @sid;
```

First the endpoint checks whether the user is already authenticated by checking whether cookies uid and pwd are provided: in this case, the user is redirected to the $index$ endpoint (that here we do not model) showing all the databases available on the website (lines 2–3). Next the session identified by cookie pre is started or a fresh one is created (lines 4–6). If the user has not sent her credentials, the page replies with a page containing the login form. This form contains a fresh CSRF token that is randomly generated for each request and stored in the session variable $\$token$. The response sent by the server contains the fresh pre-session cookie generated by the server (lines 7–10). Finally authentication is performed: a fresh session is started, a new token for CSRF

protection is generated and the user is redirected to the *index* endpoint. The response sets into the user's browser the cookies for session management and those containing the credentials. The only difference is that when login is performed via POST then the token checking is performed (lines 11–16), otherwise it is not (lines 17–21).

Now we present the encoding for the *drop* endpoint, where we let $\ell_P = (\mathsf{https}(d_P), \mathsf{https}(d_P))$. The endpoint expects three cookies: the session cookie sid and those containing the credentials stored during the login. As parameters, it expects the name of the database to be deleted (provided either via GET and POST) and the CSRF token. The encoding in our calculus follows:

1. $drop[sid, uid, pwd](g\_db, p\_db, token) \hookrightarrow$
2. **if** $@uid = \bot$ **or** $@pwd = \bot$ **then**
3. **redirect** $(login, \langle\rangle, \{\})$;
4. **start** $@sid$;
5. **if** $p\_db \neq \bot$ **then**
6. **if tokenchk**$(token, \$token)$ **then**
7. **auth** $@uid, @pwd, p\_db$ **at** ℓ_P;
8. **else**
9. **auth** $@uid, @pwd, g\_db$ **at** ℓ_P;
10. **reply** $(\{\}, \mathbf{skip}, \{\})$;

First the endpoint checks where the user is authenticated by inspecting the provided cookies: if it is not the case, the user is redirected to the *login* endpoint (lines 2–3). After starting the session identified by the cookie sid, the endpoint drops the specified database after authenticating to the DBMS using the credentials stored in the cookies: this operation is abstractly represented using the **auth** command. Like in the *login* endpoint, the CSRF token is verified when the database to be removed is provided via POST (lines 5–7) and not if sent via GET (lines 8–9).

Both endpoints are vulnerable to CSRF attacks due to the security-sensitive commands performed without any token or origin check: the **login** command in *login* on line 18 and the **auth** command in *drop* on line 9. Until December 2018, several sensitive endpoints of phpMyAdmin where vulnerable to CSRF vulnerabilities analogous to the one presented for the *drop* endpoint [MIT18a, MIT18c]. The login CSRF, instead, is a novel vulnerability that we have discovered and has been recently assigned a CVE [MIT19].

Type-checking captures the issue for the login CSRF vulnerability for the same reason of the other case studies, namely that the session cookie must be typed as low integrity and this prevents performing high integrity actions in the session. The standard CSRF is captured since it is not possible to apply rule (T-AUTH) when typing the **auth** of the *drop* endpoint in the csrf typing branch.

The fix implemented by phpMyAdmin developers is the same for both vulnerabilities, i.e., using the `$_POST` array rather than the `$_REQUEST` array to retrieve the parameters provided by the user: this ensures that all sensitive operations are performed via POST, thus the CSRF token is always checked. To model this fix in our encoding we just get read of the input variables that represent GET parameters and remove the authenticated actions involving them. The encoding of

the *login* endpoint becomes the following:

$$
\begin{array}{l}
1.\ \mathit{login}[\mathit{pre}, \mathit{uid}, \mathit{pwd}](\mathit{p\_uid}, \mathit{p\_pwd}, \mathit{token}) \hookrightarrow \\
2.\quad \textbf{if } @\mathit{uid} \neq \bot \textbf{ and } @\mathit{pwd} \neq \bot \textbf{ then} \\
3.\qquad \textbf{redirect } (\mathit{index}, \langle\rangle, \{\}); \\
4.\quad \textbf{if } @\mathit{pre} = \bot \textbf{ then} \\
5.\qquad @\mathit{pre} = \mathit{fresh}(); \\
6.\quad \textbf{start } @\mathit{pre}; \\
7.\quad \textbf{if } \mathit{p\_uid} = \bot \textbf{ then} \\
8.\qquad \$\mathit{token} = \mathit{fresh}(); \\
9.\qquad \textbf{reply } (\{\texttt{auth} \mapsto \mathsf{form}(\mathit{login}, \langle \bot, \bot, x\rangle)\}, \textbf{skip}, \{\mathit{pre} \mapsto y\}) \\
10.\qquad\quad \textbf{with } x = \$\mathit{token}, y = @\mathit{pre}; \\
11.\quad \textbf{else if tokenchk}(\mathit{token}, \$\mathit{token}) \textbf{ then} \\
12.\qquad @\mathit{sid} = \mathit{fresh}(); \textbf{login } \mathit{p\_uid}, \mathit{p\_pwd}, @\mathit{sid}; \\
13.\qquad \textbf{start } @\mathit{sid}; \$\mathit{token} = \mathit{fresh}(); \\
14.\qquad \textbf{redirect } (\mathit{index}, \langle\rangle, \{\mathit{uid} \mapsto x, \mathit{pwd} \mapsto y, \mathit{pre} \mapsto z, \mathit{sid} \mapsto z\}) \\
15.\qquad\quad \textbf{with } x = \mathit{p\_uid}, y = \mathit{p\_pwd}, z = @\mathit{sid};
\end{array}
$$

The encoding of the fixed *drop* endpoint is the following:

$$
\begin{array}{l}
1.\ \mathit{drop}[\mathit{sid}, \mathit{uid}, \mathit{pwd}](\mathit{p\_db}, \mathit{token}) \hookrightarrow \\
2.\quad \textbf{if } @\mathit{uid} = \bot \textbf{ or } @\mathit{pwd} = \bot \textbf{ then} \\
3.\qquad \textbf{redirect } (\mathit{login}, \langle\rangle, \{\}); \\
4.\quad \textbf{start } @\mathit{sid}; \\
5.\quad \textbf{if tokenchk}(\mathit{token}, \$\mathit{token}) \textbf{ then} \\
6.\qquad \textbf{auth } @\mathit{uid}, @\mathit{pwd}, \mathit{p\_db} \textbf{ at } \ell_P; \\
7.\qquad \textbf{reply } (\{\}, \textbf{skip}, \{\});
\end{array}
$$

After applying the fix, it is possible to successfully type-check our encoding of the phpMyAdmin session management logic against the web attacker.

B.2 Proof

In this section we present the full formal proof for the main result of the paper. The proof consists of two major parts: Subject Reduction ensures that typing and other invariants are preserved during execution of a web system. A relational invariant ensures that the attacked system and the unattacked system

B.2.1 Outline

In Appendix B.2.2 we introduce notation and helper functions used in the proof.

In Appendix B.2.3 we present an extended version of the semantics, containing additional annotations, as well as enriched typing rules needed to type running code. We show that the semantic rules are equivalent to the ones presented in the paper and that typing with the original rules implies typing with the extended typing rules.

In Appendix B.2.4 we prove the property of subject reduction for the system: This tells us that all components of the system are well-typed and that certain invariants are preserved during the execution of a single system.

In Appendix B.2.5 we introduce a relation between two websystems, that intuitively captures their equality on all high integrity components. We show that an attacked websystem is always in relation with its unattacked version and that this relation is preserved under execution.

In Appendix B.2.6 we combine results from the previous sections to show our main theorem.

B.2.2 Preliminaries

Here we introduce some notation that will be used in the remainder of the proof

Definition 7 (Notation)**.** *We define the following functions:*

- *For a websystem W we define servers(W) to be the set of all servers in W. For a websystem with attacker $A = (\ell_a, \mathcal{K}) \triangleright W$ we let servers(A) = servers(W)*
- *For a websystem W we define browsers(W) to be the set of all browsers in W For a websystem with attacker $A = (\ell_a, \mathcal{K}) \triangleright W$ we let browsers(A) = browsers(W)*
- *For a server $S = (D, t, \phi)$ we define urls(S) to be the set of all threads in t of the form $u[\vec{r}](\vec{x}) \hookrightarrow c$.*
- *For a server $S = (D, t, \phi)$ we define running(S) to be the set of all threads in t of the form $\lceil c \rfloor^{l,\mu}_{n^u,E}$.*
- *For a thread of the form $t = \lceil c \rfloor^{l,\mu}_{n^u,E}$ we let int$(t) = l$*
- *For a thread of the form $t = \lceil c \rfloor^{l,\mu}_{n^u,i,j}$ and a database of global memories $D_@$ we let mem$_g(D_@, t) = D_@(i)$.*

- *For a thread of the form* $t = \lceil c \rfloor^{l,\mu}_{n^u,i,j}$ *and a database of session memories* $D_\$$ *we let* $mem_s(D_\$, t) = D_\$(j)$.
- *For a reference type* $\tau_r = \texttt{ref}(\tau)$ *we let* $ref_\tau(\tau_r) = \tau$.
- *For a command* c *we let* $coms(c)$ *be the set containing all commands in* c.
- *For an event* $\alpha@l$ *we define* $sync_I(\alpha@l) = l$ *as the sync integrity of the event*
- *We define a meet* $\overline{\sqcap}$ *between a type* τ *and a label* ℓ *that limits the label of* τ *to the label* ℓ. *Formally:*
$$\tau \overline{\sqcap} \ell = \begin{cases} \ell' \sqcap \ell & \text{if } \tau = \ell' \\ \texttt{cred}(\ell \sqcap \ell') & \text{if } \tau = \texttt{cred}(\ell') \end{cases}$$
- *We define a join* $\tilde{\sqcup}$ *on types that behaves like the regular join* $\tau_1 \sqcup \tau_2$ *if it is defined and* $label(\tau_1) \sqcup label(\tau_2)$ *otherwise*
- *We define a join* $\tilde{\sqcup}_I$ *as* $\texttt{cred}(\ell)\tilde{\sqcup}_I l := \texttt{cred}((C(\tau), I(\tau) \sqcup_I l))$ *and* $\ell \tilde{\sqcup}_I l := (C(\tau), I(\tau) \sqcup_I l)$
- *For a running server thread* $t = \lceil c \rfloor^{l,\mu}_{E,R}$ *we let* $int_\sqcup(t) = l \sqcup_I \bigsqcup_{I l' \in \{l' \mid \text{reset } l' \in c\}} l'$
- *For value* v^τ, *We define* $jlabel(v^\tau) = (C(\tau) \sqsubseteq_C C(\ell_a)) \ : \ (\bot_C, \top_I) \ ? \ label(\tau)$
- *For a typing environment* Γ *and two memories* M *and* M', *we write* $M =_{\Gamma,\bot_I} M'$ *if for all* r *with* $I(\ell_a) \not\sqsubseteq_I I(ref_\tau(\Gamma(r)))$ *we have* $M(r) = M'(r)$
- *For a set of name* N, *we let* $\lfloor N \rfloor_{\ell_a}$ *be the set same set of names, where all types have been lowered to* ℓ_a.

Definition 8 (Freshness)**.**
- *A Browser* $B = (N, K, P, T, Q, \vec{a})^{\iota_b}$ *is fresh if* $N = \{\}$, $K = \{\}$, $P = \{\}$, $T = \{\}$, $Q = \{\}$.
- *A Server* $S = (D, \Phi, t)$ *is fresh if* $D = \{\}$ *and* $\Phi = \{\}$. *(also see Definition 5)*
- *A Websystem* W *is fresh if all* $B \in browsers(W)$ *and all* $S \in servers(W)$ *are fresh.*

B.2.3 Extended Semantics and Typing Rules

In this section we introduce additional and modified rules for the semantics and the type system. In Lemma 1, we then prove that this semantic is equivalent to the original semantics for well-typed systems.

The most important changes are presented here:

- We annotate running server threads, the browser state, the DOM, and network requests and replies with an integrity label $l \in \mathcal{L}$ and an attacked state $\mu \in \{\texttt{hon}, \texttt{att}\}$. Intuitively, l is dynamically tracking which domains have influenced the current state of the execution, while μ is a binary flag that tells us whether the attacker used his capabilities to directly influence the current state.

- We annotate events with an integrity label (an additional one, using the notation $\alpha@l$) . This label is used to synchronize the execution of the unattacked and the attacked websystem in the relation: High integrity events have to be processed in sync, while low integrity events may be processed individually.

- We introduce a new command reset l for servers to "reset the pc" after a conditional. This operation has no semantic effect, it just updates the integrity annotation .

- We partition the database $D = (D_@, D_\$)$ into two different mappings for global and server memories.

- We split the rule (A-TIMEOUT) into two separate rules (A-TIMEOUTSEND) and (A-TIMEOUTRECV). We therefore introduce a buffer in the network state that keeps track of open connections that require a response. This is required since in the relation proof, every request and response needs to be atomic, so that it can be matched with the corresponding request or response in the other system. For example a request to a low integrity domain, that is intercepted by the attacker might be processed using a timeout in the unattacked system.

- All values $v^\tau \in \mathcal{V}$ (in the code, in the DOM, in memory or in requests and responses) are now annotated with a security type that gives us runtime information. All primitive values have by default the type $\tau = \bot$ and hence can be given any security label ℓ (due to subtyping). Since for names $n^\tau \in \mathcal{N}$ we have $\tau = \texttt{cred}(\ell)$ for some ℓ, we cannot use subtyping if $C(\tau) \not\sqsubseteq_C C(\ell_a)$ or $I(\ell_a) \not\sqsubseteq_I I(\tau)$. We hence partition the set of names $\mathcal{N} = \mathcal{N}_0 \uplus_{C(\ell) \not\sqsubseteq_C C(\ell_a), \vee I(\ell_a) \not\sqsubseteq_I I(\ell)} \mathcal{N}_\ell$ into one set $\mathcal{N}_0$ of names of low confidentiality and integrity and one set $\mathcal{N}_\ell$ for each label ℓ with high confidentiality or integrity.

We define a translation $\bar{\cdot}$ function from a fresh websystem in the original semantics to websystems in the extended semantics.

Intuitively, the translation annotates all constants with the type $\bot$ and lets the initial browser start with high integrity and in the honest mode.

$$
\begin{array}{rcll}
\overline{(\ell_a, \mathcal{K}) \triangleright W} & = & (\ell_a, \mathcal{K}) \triangleright_{\emptyset} \overline{W} & \\
\overline{W \parallel W'} & = & \overline{W} \parallel \overline{W'} & \\
\overline{(\{\}, \{\}, t)} & = & ((\{\}, \{\}), \{\}, \bar{t}) & \\
\overline{t \parallel t'} & = & \bar{t} \parallel \overline{t'} & \\
\overline{u[\vec{r}](\vec{x}) \hookrightarrow c} & = & u[\vec{r}](\vec{x}) \hookrightarrow \bar{c} & \\
\overline{\mathbf{skip}} & = & \mathbf{skip} & \\
\overline{c; c;} & = & \bar{c}; \overline{c'} & \\
\overline{@r := se} & = & @r := \overline{se} & \\
\overline{\$r := se} & = & \$r := \overline{se} & \\
\overline{\mathbf{if}\ se\ \mathbf{then}\ c\ \mathbf{else}\ c'} & = & \mathbf{if}\ \overline{se}\ \mathbf{then}\ \bar{c}\ \mathbf{else}\ \overline{c'} & \\
\overline{\mathbf{login}\ se_u, se_{pw}, se_{id}} & = & \mathbf{login}\ \overline{se_u}, \overline{se_{pw}}, \overline{se_{id}} & \\
\overline{\mathbf{start}\ se} & = & \mathbf{start}\ \overline{se} & \\
\overline{\mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell} & = & \mathbf{auth}\ \overline{\vec{se}}\ \mathbf{at}\ \ell & \\
\overline{\mathbf{if\ tokenchk}(e, e')\ \mathbf{then}\ c} & = & \mathbf{if\ tokenchk}(\bar{e}, \overline{e'})\ \mathbf{then}\ \bar{c} & \\
\overline{\mathbf{if\ originchk}(L)\ \mathbf{then}\ c} & = & \mathbf{if\ originchk}(L)\ \mathbf{then}\ \bar{c} & \\
\overline{\mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se}} & = & \mathbf{reply}\ (\overline{page}, \bar{s}, \overline{ck})\ \mathbf{with}\ \vec{x} = \overline{\vec{se}} & \\
\overline{\mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se}} & = & \mathbf{redirect}\ (\bar{u}, \overline{\vec{z}}, \overline{ck})\ \mathbf{with}\ \vec{x} = \overline{\vec{se}} & \\
\overline{x} & = & x & \\
\overline{@r} & = & @r & \\
\overline{\$r} & = & \$r & \\
\overline{\mathit{fresh}()^{\ell}} & = & \mathit{fresh}()^{\ell} & \\
\overline{se \odot se'} & = & \overline{se} \odot \overline{se'} & \\
\overline{v} & = & v^{\perp} & \text{with } v \notin \mathcal{N} \\
\overline{n^{\ell}} & = & n^{\mathtt{cred}(\ell)} & \\
\overline{\perp} & = & \perp & \\
\overline{(\{\}, \{\}, \{\}, \{\}, \{\}, \vec{a})^{\iota_b}} & = & (\{\}, \{r \mapsto \perp^{\Gamma_{\mathcal{R}^{@}(r)}}\}, \{\}, \{\}, \{\}, \vec{a})^{\iota_b, \perp_I, \mathsf{hon}} &
\end{array}
$$

The extended semantics are presented in Table B.6, Table B.7 and Table B.8. As a convention, we use $\stackrel{\alpha}{\Rightarrow}$ for steps derived using the extended semantics and $\stackrel{\alpha}{\rightarrow}$ for steps derived using the original semantics.

Detailed explanation of extended browser semantics

- The definition of $upd\_ck'(\cdot,\cdot,\cdot)$ is like the original definition of $upd\_ck(\cdot,\cdot,\cdot)$, with the difference that the type annotations of values are joined with the type of the reference in the environment Γ. We will show in the proof that typing then ensures that the types of values in a memory reference is always equal to the type for that reference in the environment Γ.
- (BE-VAL) simply adds the value type
- (BE-BINOP) adds types, and assigns the join of the labels of the input types to the result.

$upd\_ck'(M,u,ck) = M \triangleleft (ck \uparrow' u)$ where $ck \uparrow' u$ is the map ck' such that $ck'(r) = v^{\tau \sqcup \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@(r)})}$ iff $ck(r) = v^\tau$ and $I(\lambda(u)) \sqsubseteq_I I(\lambda(r))$.

Expressions

(BE-VAL)

$$\frac{}{eval_\ell(v^\tau, M, f) = v^\tau}$$

(BE-BINOP)

$$\frac{eval_\ell(be, M, f) = v^\tau \qquad eval_\ell(be', M, f) = v'^{\tau'}}{eval_\ell(be \odot be', M, f) = (v \odot v')^{label(\tau) \sqcup label(\tau')}}$$

(BE-REFERENCE)

$$\frac{C(\lambda(r)) \sqsubseteq_C C(\ell)}{eval_\ell(r, M, f) = M(r)}$$

(BE-DOM)

$$\frac{\begin{array}{c} eval_\ell(be, M, f) = v^\tau \qquad eval_\ell(be', M, f) = v'^{\tau'} \\ \{v \mapsto \mathsf{form}(u^{\tau_u}, \vec{v^\tau})\} \in f \qquad v' = 0 \Rightarrow v''^{\tau''} = u^{\tau_u} \qquad v' \neq 0 \Rightarrow v''^{\tau''} = v^{\tau_{v'}}_{v'} \end{array}}{eval_\ell(\mathsf{dom}(be, be'), M, f) = v''^{\tau'' \sqcup_I I(\tau) \sqcup_I I(\tau')}}$$

Browser

(B-LOAD)

$$\frac{\begin{array}{c} n \leftarrow \mathcal{N} \qquad \alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, \bot)^{I(\lambda(u)),\mathsf{hon}} \\ ck = get\_ck(M,u) \qquad (\mathrm{orig}(u) = \mathsf{http}(d) \Rightarrow d \notin \Delta) \end{array}}{\begin{array}{c} (\{\}, M, P, \{\}, \{\}, \mathsf{load}(tab, u, \sigma) :: \vec{a})^{\iota_b, \bot_I, \mathsf{hon}} \xrightarrow[\Gamma]{\bullet @ \bot_I} \\ (\{n \mapsto (tab, u, \bot)\}, M, P, \{\}, \{\alpha @ \bot_I\}, \vec{a})^{\iota_b, \lambda(u), \mathsf{hon}} \end{array}}$$

(B-INCLUDE)

$$\frac{\begin{array}{c} n \leftarrow \mathcal{N} \qquad ck = get\_ck(M,u) \\ \{tab \mapsto (u', f, l'\mu')\} \in P \qquad \forall k \in [1 \ldots |\vec{be}|] : p(k) = eval_{\lambda(u')}(be_k, M, f) \\ \alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, \mathrm{orig}(u'))^{l \sqcup_I I(\lambda(u)), \mu} \qquad (\mathrm{orig}(u) = \mathsf{http}(d) \Rightarrow d \notin \Delta) \end{array}}{\begin{array}{c} (\{\}, M, P, \{tab \mapsto \mathbf{include}(u, \vec{be})\}, \{\}, \vec{a})^{\iota_b, l, \mu} \xrightarrow[\Gamma]{\bullet @ l} \\ (\{n \mapsto (tab, u, \mathrm{orig}(u'))\}, M, P, \{tab \mapsto \mathbf{skip}\}, \{\alpha @ l\}, \vec{a})^{\iota_b, l, \mu} \end{array}}$$

(B-SUBMIT)

$$\frac{\begin{array}{c} \alpha = \overline{\mathsf{req}}(\iota_b, n, u', p, ck, \mathrm{orig}(u))^{l' \sqcup_I I(\lambda(u')), \mu'} \\ \{tab \mapsto (u, f, l', \mu')\} \in P \qquad \{v' \mapsto \mathsf{form}(u'^\tau, \vec{v^\tau})\} \in f \\ \forall k \in [1 \ldots |\vec{v}|].\, p(k) = k \in dom(p)\ ?\ p(k) : v^{\tau_k}_k \qquad n \leftarrow \mathcal{N} \qquad ck = get\_ck(M,u) \end{array}}{\begin{array}{c} (\{\}, M, P, \{\}, \{\}, \mathsf{submit}(tab, u, v', p) :: \vec{a})^{\iota_b, \bot_I, \mathsf{hon}} \xrightarrow[\Gamma]{\bullet @ \bot_I} \\ (\{n \mapsto (tab, u', \mathrm{orig}(u))\}, M, P, \{\}, \{\alpha @ \bot_I\}, \vec{a})^{\iota_b, l' \sqcup_I I(\lambda(u)), \mu'} \end{array}}$$

Table B.6: Extended semantics of browsers.

Browser (continued)

(B-RECVLOAD)

$$\frac{\begin{array}{c}M' = upd\_ck'(M,u,ck)\\ \alpha = \mathsf{res}(\iota_b,n,u,\bot,\{\},ck,page,s)^{l',\mu'} \qquad \vec{a}\,' = (page = \mathsf{error} \wedge \iota_b = \mathsf{usr})\ ?\ (\mathsf{halt} :: \vec{a}) : \vec{a}\end{array}}{\begin{array}{c}(\{n \mapsto (tab,u,o)\},M,P,\{\},\{\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{\alpha@l}_{\Gamma}\\ (\{\},M',P \triangleleft \{tab \mapsto (u,page,l',\mu')\},\{tab \mapsto s\},\{\},\vec{a}\,')^{\iota_b,l',\mu'}\end{array}}$$

(B-RECVINCLUDE)

$$\frac{\begin{array}{c}M' = upd\_ck'(M,u,ck)\\ \alpha = \mathsf{res}(\iota_b,n,u,\bot,\{\},ck,page,s)^{l',\mu'} \qquad \mu'' = (\mu = \mathsf{att} \vee \mu' = \mathsf{att})\ ?\ \mathsf{att}\ :\ \mathsf{hon}\end{array}}{\begin{array}{c}(\{n \mapsto (tab,u,o)\},M,P,\{tab \mapsto s'\},\{\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{@l \sqcap_I l'}_{\Gamma}\\ (\{\},M',P,\{tab \mapsto s;s'\},\{\},\vec{a})^{\iota_b,l' \sqcup_I l,\mu''}\end{array}}$$

(B-REDIRECT)

$$\frac{\begin{array}{c}\alpha = \mathsf{res}(n,u,u',\vec{v},ck,\bot,\bot,\bot)^{l',\mu'} \qquad M' = upd\_ck'(M,u,ck) \qquad n' \leftarrow \mathcal{N}\\ ck' = get\_ck(M',u') \qquad \forall k \in [1\ldots|\vec{v}|] : p(k) = v_k \qquad o' = (o = \mathrm{orig}(u))\ ?\ o : \bot\\ \alpha' = \overline{\mathsf{req}}(\iota_b,n',u',p,ck',o')^{l',\mu'} \qquad (\mathrm{orig}(u') = \mathsf{http}(d) \Rightarrow d \notin \Delta)\end{array}}{\begin{array}{c}(\{n \mapsto (tab,u,o)\},M,P,T,\{\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{\alpha@l \sqcap_I l'}_{\Gamma}\\ (\{n' \mapsto (tab,u',o')\},M',P,T,\{\alpha'@l'\},\vec{a})^{\iota_b,l',\mu'}\end{array}}$$

(B-FLUSH)

$$\frac{}{(N,M,P,T,\{\alpha@l'\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{\alpha@l'}_{\Gamma} (N,M,P,T,\{\},\vec{a})^{\iota_b,l,\mu}}$$

(B-END)

$$\frac{}{(\{\},M,P,\{tab \mapsto \mathbf{skip}\},\{\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{\bullet@\bot_I}_{\Gamma} (\{\},M,P,\{\},\{\},\vec{a})^{\iota_b,\bot_I,\mathsf{hon}}}$$

(B-SEQ)

$$\frac{(\{\},M,P,\{tab \mapsto s\},\{\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{\alpha@l}_{\Gamma} (\{\},M',P',\{tab \mapsto s'\},\{\},\vec{a})^{\iota_b,l',\mu'}}{(\{\},M,P,\{tab \mapsto s;s''\},\{\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{\alpha@l}_{\Gamma} (\{\},M',P',\{tab \mapsto s';s''\},\{\},\vec{a})^{\iota_b,l',\mu'}}$$

(B-SKIP)

$$\frac{}{(\{\},M,P,\{tab \mapsto \mathbf{skip};s\},\{\},\vec{a})^{\iota_b,l,\mu} \xRightarrow{\bullet@l}_{\Gamma} (\{\},M,P,\{tab \mapsto s\},\{\},\vec{a})^{\iota_b,l,\mu}}$$

Table B.6: Extended semantics of browsers (continued).

Browser (continued)

(B-SETREFERENCE)

$$\frac{\{tab \mapsto (u, f, l', \mu')\} \in P \qquad \ell = \lambda(u) \qquad eval_\ell(be, M, f) = v^\tau \qquad I(\ell) \sqsubseteq_I I(\lambda(r))}{(\{\}, M, P, \{tab \mapsto r := be\}, \{\}, \vec{a})^{\iota_b, l, \mu} \xrightarrow[\Gamma]{\bullet@l} (\{\}, M\{r \mapsto v^{\tau \sqcup \mathsf{ref}_\tau(\Gamma_{\mathcal{R}@}(r))}\}, P, \{tab \mapsto \mathbf{skip}\}, \{\}, \vec{a})^{\iota_b, l, \mu}}$$

(B-SETDOM)

$$\frac{\begin{array}{c}\ell = \lambda(u') \qquad \{tab \mapsto (u', f, l', \mu')\} \in P \\ eval_\ell(be', M, f) = v' \qquad \forall k \in [1 \ldots |\vec{be}|].\, v_k'^{\tau'} = eval_\ell(be_k, M, f) \wedge v_k^\tau = v_k'^{\tau' \sqcup_I l} \\ \mu'' = (\mu = \mathsf{att} \vee \mu' = \mathsf{att}) \;?\; \mathsf{att} \;:\; \mathsf{hon}\end{array}}{(\{\}, M, P \uplus \{tab \mapsto (u', f\}, \{tab \mapsto \mathbf{setdom}(be', u, \vec{be})\}, \{\}, \vec{a})^{\iota_b, l, \mu} \xrightarrow[\Gamma]{\bullet@l} (\{\}, M, P \uplus \{tab \mapsto (u', f\{v' \mapsto \mathsf{form}(u^{(\bot_C, l)}, \vec{v^\tau})\}, l' \sqcup_I l, \mu'')\}, \{tab \mapsto \mathbf{skip}\}, \{\}, \vec{a})^{\iota_b, l, \mu}}$$

Table B.6: Extended semantics of browsers (continued).

- (BE-REFERENCE) is the same as in the original semantics.
- (BE-DOM) adds types. The integrity label of the returned value is lowered, taking into account the integrity labels of the two parameters.
- (B-LOAD) adds the integrity label of the URL and the hon flag to the request. Additionally, the request is marked as a high integrity sync action. This means that all load events have to be processed in sync between the attacked and unattacked system. The integrity label of the browser state the integrity label of the URL and the attacked mode is honest.
- (B-INCLUDE) uses the join of the browser integrity label and the URL's integrity label, as well as the browser's current attack state as annotations on the request. The event's sync integrity label is the browser's integrity label. The rule does not modify the browser's integrity label or attacked state.
- (B-SUBMIT) uses the integrity label and attacked mode from the DOM for the request, combined with the integrity label of the target URL The rule does not modify the browser's integrity label or attacked state.
- (B-RECVLOAD) receives a response to a load event, labelled with an integrity label and an attacked state, and uses these labels for the DOM and the browser state. The sync integrity of the response event is the integrity label of the browser. This means that event direct responses to a load or a submit have to be processed in sync (since they leave the browser in a high integrity state). A redirect however can lower the integrity of a browser that is awaiting a response to a load or submit (see below).

- (B-RECVINCLUDE) joins the integrity label and attacked state of the current browser state with the ones from the network response and uses them in the continuation. The sync integrity label of the event is the meet of the integrity label of the reply and the integrity label of the browser. This means that as long as one of the two is high, the response to the include has to be processed in sync.

- (B-REDIRECT) uses the integrity label and attacked state of the incoming event for the outgoing event and the resulting browser state. The sync integrity label of the event is the meet of the integrity label of the reply and the integrity label of the browser. This means that as long as one of the two is high, the response to the include has to be processed in sync.

- (B-FLUSH) sends out the event from the buffer together with its sync integrity label.

- (B-END) resets the browser's integrity label to high integrity and resets the attacked mode to hon. The sync integrity label is high, meaning that this step always has to be processed in sync.

- (B-SEQ) propagates the labels from the subcommand.

- (B-SKIP) propagates the browser annotations. The sync integrity label is the integrity label of the browser state

- (B-SETREFERENCE) evaluates the expression and stores it in the memory, with the join of computed type and the type of the reference in the typing environment Γ.

- (B-SETDOM) updates the DOM labelling by joining its original integrity label and the attacked sate with the ones of the browser state. The integrity label of the value stored into the DOM is lowered using the integrity label of the browser.

Detailed explanation of extended server semantics

- (SE-VAL) also contains the type.

- (SE-FRESH) samples names from the partition of the set of names indicated by the annotation. If the browser id is not the one of the honest user usr, then we always sample from $\mathcal{N}_0$, the set of names of low confidentiality and integrity.

- (SE-BINOP) is just like (BE-BINOP)

- (SE-READGLOABL), (SE-READSESSION) look up the reference in the corresponding part of the database.

- (S-SEQ) just propagates the annotations

Expressions

(SE-VAL)

$$\frac{}{eval_E(v^\tau, D) = v^\tau}$$

(SE-FRESH)

$$\frac{\begin{array}{c}\iota_b = \mathsf{usr} \Rightarrow n^{\tau'} \leftarrow \mathcal{N}_\tau \\ \iota_b \neq \mathsf{usr} \Rightarrow n^{\tau'} \leftarrow \mathcal{N}_0\end{array}}{eval_E(\mathit{fresh}()^\tau, D) = n^{\tau'}}$$

(SE-BINOP)

$$\frac{eval_E(se, D) = v^\tau \; eval_E(se', D) = v'^{\tau'}}{eval_E(se \odot se', D) = (v \odot v')^{label(\tau) \sqcup label(\tau')}}$$

(SE-READGLOBAL)

$$\frac{}{eval_{i,\_}(@r, (D_@, D_\$)) = D_@(i, r)}$$

(SE-READSESSION)

$$\frac{}{eval_{i,j}(\$r, (D_@, D_\$)) = D_\$(j, r)}$$

Server

(S-SEQ)

$$\frac{(D, \phi, \lceil c \rfloor^{l,\mu}_{R,E}) \overset{\alpha@l}{\Longrightarrow}_\Gamma (D', \phi', \lceil c' \rfloor^{l',\mu}_{R,E'})}{(D, \phi, \lceil c; c'' \rfloor^{l,\mu}_{R,E}) \overset{\alpha@l}{\Longrightarrow}_\Gamma (D', \phi', \lceil c'; c'' \rfloor^{l',\mu}_{R,E'})}$$

(S-IFTRUE)

$$\frac{\begin{array}{c}c'' = (\mathbf{reply}, \mathbf{redir}, \mathbf{tokencheck}, \mathbf{origincheck} \in coms(c)) \; ? \; c \; : \; c; \mathsf{reset}\; l \\ eval_E(se, D) = true^\tau \qquad l' = l \sqcup_I I(\tau)\end{array}}{(D, \phi, \lceil \mathbf{if}\ se\ \mathbf{then}\ c\ \mathbf{else}\ c' \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l}{\Longrightarrow}_\Gamma (D, \phi, \lceil c'' \rfloor^{l',\mu}_{R,E})}$$

(S-IFFALSE)

$$\frac{\begin{array}{c}c'' = (\mathbf{reply}, \mathbf{redir}, \mathbf{tokencheck}, \mathbf{origincheck} \in coms(c)) \; ? \; c \; : \; c; \mathsf{reset}\; l \\ eval_E(se, D) = false^\tau \qquad l' = l \sqcup_I I(\tau)\end{array}}{\begin{array}{c}(D, \phi, \lceil \mathbf{if}\ se\ \mathbf{then}\ c\ \mathbf{else}\ c' \rfloor^{l,\mu}_{R,E}) \\ \overset{\bullet@l}{\Longrightarrow}_\Gamma (D, \phi, \lceil c'' \rfloor^{l',\mu}_{R,E})\end{array}}$$

(S-RESET)

$$\frac{}{\begin{array}{c}(D, \phi, \lceil \mathsf{reset}\; l' \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l'}{\Longrightarrow}_\Gamma \\ (D, \phi, \lceil \mathbf{skip} \rfloor^{l',\mu}_{R,E})\end{array}}$$

(S-SKIP)

$$\frac{}{(D, \phi, \lceil \mathbf{skip}; c \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l}{\Longrightarrow}_\Gamma (D, \phi, \lceil c \rfloor^{l,\mu}_{R,E})}$$

Table B.7: Extended semantics of server.

Server (continued)

(S-TCTRUE)
$$\frac{eval_E(se, D) = v^\tau \qquad eval_E(se', D) = v'^{\tau'} \qquad v = v'}{(D, \phi, \lceil \textbf{if tokenchk}(se, se') \textbf{ then } c \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l}{\Longrightarrow}_\Gamma (D, \phi, \lceil c \rfloor^{l,\mu}_{R,E})}$$

(S-TCFALSE)
$$\frac{eval_E(se, D) = v^\tau \qquad eval_E(se', D) = v'^{\tau'} \qquad v \neq v'}{(D, \phi, \lceil \textbf{if tokenchk}(se, se') \textbf{ then } c \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l}{\Longrightarrow}_\Gamma (D, \phi, \lceil \textbf{reply } (\mathsf{error}, \textbf{skip}, \{\}) \rfloor^{l,\mu}_{R,E})}$$

(S-RECV)
$$\frac{\begin{array}{c}\alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{l,\mu} \qquad i \leftarrow \mathcal{N} \\ \forall k \in [1 \ldots |\vec{r}|].\, M(r_k) = (r_k \in \mathit{dom}(ck)) \; ? \; ck(r_k) : \bot \\ m = |\vec{x}| \qquad \forall k \in [1 \ldots \mu].\, v_k = (k \in \mathit{dom}(p)) \; ? \; p(k) : \bot \\ \sigma = [x_1 \mapsto v_1, \ldots, x_m \mapsto v_m]\end{array}}{\begin{array}{c}(D, \phi, u[\vec{r}](\vec{x}) \hookrightarrow c) \overset{\alpha@l}{\Longrightarrow}_\Gamma \\ (D \uplus \{i \mapsto M\}, \phi, \lceil c\sigma \rfloor^{l,\mu}_{(n,u,\iota_b,o),(i,\bot)} \parallel u[\vec{r}](\vec{x}) \hookrightarrow c)\end{array}}$$

(S-RESTORESESSION)
$$\frac{E = i, \_ \qquad eval_E(se, D) = j^\tau \qquad j \in \mathit{dom}(D)}{(D, \phi, \lceil \textbf{start } se \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l}{\Longrightarrow}_\Gamma (D, \phi, \lceil \textbf{skip} \rfloor^{l,\mu}_{R,i,j})}$$

(S-NEWSESSION)
$$\frac{\begin{array}{c}E = i, \_ \qquad eval_E(se, D) = j^\tau \\ j \notin \mathit{dom}(D)\end{array}}{(D, \phi, \lceil \textbf{start } se \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l}{\Longrightarrow}_\Gamma (D \uplus \{j \mapsto \{r \mapsto \bot^{\Gamma_{\mathcal{R}^\$}(r) \tilde{\sqcup} jlabel(j)}\}\}, \phi, \lceil \textbf{skip} \rfloor^{l,\mu}_{R,i,j})}$$

(S-OCHKFAIL)
$$\frac{R = n, u, \iota_b, o \qquad o \notin O}{\begin{array}{c}(D, \phi, \lceil \textbf{if originchk}(O) \textbf{ then } c \rfloor^{l,\mu}_{R,E}) \overset{\bullet}{\Rightarrow}_\Gamma \\ (D, \phi, \lceil \textbf{reply } (\mathsf{error}, \textbf{skip}, \{\}) \rfloor^{l,\mu}_{R,E})\end{array}}$$

(S-OCHKSUCC)
$$\frac{R = n, u, \iota_b, o \qquad o \in L}{\begin{array}{c}(D, \phi, \lceil \textbf{if originchk}(L) \textbf{ then } c \rfloor^{l,\mu}_{R,E}) \overset{\bullet@l}{\Longrightarrow}_\Gamma \\ (D, \phi, \lceil c \rfloor^{l,\mu}_{R,E})\end{array}}$$

Table B.7: Extended semantics of server (continued).

Server (continued)

(S-SETGLOBAL)

$$\frac{eval_E(se, D) = v^\tau \qquad D = (D_@, D_\$) \qquad \Gamma'_{\mathcal{R}^@} = (\iota_b = \mathsf{usr})\ ?\ \Gamma_{\mathcal{R}^@}\ :\ \{\_ \mapsto \ell_a\} \qquad \tau' = \tau \tilde{\sqcup}_I l}{(D, \phi, \lceil @r := se \rfloor^{l,\mu}_{R,(i,j)}) \xrightarrow[\Gamma]{\bullet @l} (D\{i \mapsto D_@(i)\{r \mapsto v^{\tau'}\}\}, \phi, \lceil \mathbf{skip} \rfloor^{l,\mu}_{R,(i,j)})}$$

(S-SETSESSION)

$$\frac{eval_{i,j}(se, D) = v \qquad D = (D_@, D_\$) \qquad \tau' = \tau \sqcup (\mathrm{ref}_\tau(\Gamma_{\mathcal{R}^\$}(r)) \tilde{\sqcup} jlabel(j))}{(D, \phi, \lceil \$r := se \rfloor^{l,\mu}_{R,(i,j)}) \xrightarrow[\Gamma]{\bullet @l} (D\{j \mapsto D_\$(j)\{r \mapsto v^{\tau'}\}\}, \phi, \lceil \mathbf{skip} \rfloor^{l,\mu}_{R,i,j})}$$

(S-LOGIN)

$$\frac{eval_E(se_{usr}, D) = \iota_s \qquad eval_E(se_{pw}, D) = \rho(\iota_s, u) \qquad eval_E(se_{sid}, D) = n}{(D, \phi, \lceil \mathbf{login}\ se_{usr}, se_{pw}, se_{sid} \rfloor^{l,\mu}_{R,E}) \xrightarrow[\Gamma]{\bullet @l} (D, \phi \triangleleft \{n \mapsto \iota_s\}, \lceil \mathbf{skip} \rfloor^{l,\mu}_{R,E})}$$

(S-AUTH)

$$\frac{R = n, u, \iota_b, o \qquad j \in dom(\phi) \qquad \forall k \in [1 \ldots |\vec{se}|].\ eval_{i,j}(se_k, D) = v_k}{(D, \phi, \lceil \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell \rfloor^{l,\mu}_{R,i,j}) \xrightarrow[\Gamma]{\sharp[\vec{v}]^{\iota_b,\iota_s} @ l} (D, \phi, \lceil \mathbf{skip} \rfloor^{l,\mu}_{R,i,j})}$$

(S-REPLY)

$$\frac{\begin{array}{c} R = n, u, \iota_b, o \\ \Gamma_{\mathcal{U}}(u) = \_, \_, l_r \qquad m = |\vec{x}| = |\vec{se}| \qquad \forall k \in [1, m].\ eval_E(se_k, D) = v_k \\ \sigma = [x_1 \mapsto v_1, \ldots, x_m \mapsto v_m] \qquad \alpha = \overline{\mathsf{res}}(\iota_b, n, u, \bot, \{\}, ck\sigma, page\sigma, s\sigma)^{l \sqcup_I l_r, \mu} \\ c' = (page = error)\ ?\ \mathbf{bad}\ :\ \mathbf{halt} \end{array}}{(D, \phi, \lceil \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se} \rfloor^{l,\mu}_{R,E}) \xrightarrow[\Gamma]{\alpha @ l} (D, \phi, \lceil c' \rfloor^{l,\mu}_{R,E})}$$

(S-LPARALLEL)

$$\frac{(D, \phi, t) \xrightarrow[\Gamma]{\alpha @ l} (D', \phi', t'')}{(D, \phi, t \parallel t') \xrightarrow[\Gamma]{\alpha @ l} (D', \phi', t'' \parallel t')}$$

Table B.7: Extended semantics of server (continued).

Server (continued)

(S-REDIRECT)

$$\frac{\begin{array}{c} R = n, u, \iota_b, o \\ \Gamma_{\mathcal{U}}(u) = \_, \_, l_r \qquad m = |\vec{x}| = |\vec{se}| \qquad \forall k \in [1, m].\ eval_E(se_k, D) = v_k \\ \sigma = [x_1 \mapsto v_1, \ldots, x_m \mapsto v_m] \qquad \alpha = \overline{\mathsf{res}}(\iota_b, n, u, u', \vec{z}\sigma, ck\sigma, \{\}, \mathbf{skip})^{l \sqcup_I l_r, \mu} \end{array}}{(D, \phi, \lceil \mathbf{redirect}\ (u', \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se} \rfloor^{l,\mu}_{R,E}) \xRightarrow[\Gamma]{\alpha@l} (D, \phi, \lceil \mathbf{halt} \rfloor^{l,\mu}_{R,E})}$$

(S-RPARALLEL)

$$\frac{(D, \phi, t') \xRightarrow[\Gamma]{\alpha@l} (D', \phi', t'')}{(D, \phi, t \parallel t') \xRightarrow[\Gamma]{\alpha@l} (D', \phi', t \parallel t'')}$$

Table B.7: Extended semantics of server (continued).

- (S-IFTRUE), (S-IFFALSE) lower the integrity label, based on the type of the guard. In case the code for the branch does not contain any command that can lead to a response, a reset command is added after the branch, to bring the integrity label back to its original value.
- (S-RESET) restores the integrity label to the provided value. The sync integrity label is the integrity label to which the reset is performed. This means that returning to a high integrity context from a low integrity context must be processed in sync.
- (S-SKIP) just propagates the annotations
- (S-TCTRUE), (S-TCFALSE) just propagate the annotations.
- (S-RECV) takes the annotations from the request and uses them for the newly started thread.
- (S-RESTORESESSION) just propagates the annotations.
- (S-NEWSESSION) initializes the new memory with $\perp$, annotated with the appropriate type from $\Gamma_{\mathcal{R}^\$}$ combined with the type of the session identifier. The integrity label is not influenced, as by an invariant the integrity of all session memory references and the user identity is upper bounded by the integrity of the session identifier
- (S-OCHCKSUCC), (S-OCHCKFAIL) just propagate the annotations.
- (S=LPARALLEL), (R-PARALLEL) juts propagate the labelling of the events of sub threads
- (S-SETGLOBAL) stores the value with its computed type, joining the integrity label with the thread's integrity label.

- (S-SETSESSION) stores the value with the type that results from joining the value's original type with the type of the reference, limited by the type of the session identifier. We will show in the proof that typing then ensures that the types of values in a memory reference is always equal to the type for that reference in the environment Γ, limited by the type of the session identifier.

- (S-LOGIN) just propagates the annotations

- (S-AUTH) just propagates the annotations

- (S-REPLY), uses the annotations of the current thread for the reply, where the integrity label is joined with the expected integrity label for the reply. In case the reply is an error message, instead of going to the regular **halt** state, the thread will go to a **bad** state. These two states are semantically equivalent (both cannot be processed further) and are just used to establish an invariant in the proofs.

- (S-REDIRECT) uses the annotations of the current thread for the reply where again the integrity label is joined with the expected integrity label for the reply.

Detailed explanation of extended semantics of seb systems with the attacker

For the proof it is required that every rule only performs a single step in a browser. We hence have to split up the rule (A-TIMEOUT) into two separate rules. For this reason we introduce a buffer T_O that stores the request that requires the timeout-response. As long as this buffer contains an element, the only rule that can be taken is (A-TIMEOUTRECV).

- (W-LPARALLEL), (W-RPARALLEL) and

- (A-NIL) simply propagate the annotations.

- (A-BROWSERSERVER) "forwards" the request with the same annotations. We use the sync label of the browser event for the event in the websystem and use the integrity label of the browser event as the sync label for the server event. This means that in some cases (for example for a load to a URL of low integrity) we will require that the browser step is performed in sync, while the server step must not be in sync, we just require that the request is processed in some form. For example, it is possible to match a server receiving a low integrity request with a case where the attacker interferes.

- (A-SERVERBROWSER) does the same in the other direction. Again we use the browser event's sync integrity label for the websystem event. This allows us to synchronize two browsers receiving a low integrity a response to a load request with high sync integrity label, without synchronizing the server step. For example we can match a server responding to the request with the attacker responding to the request.

(W-LPARALLEL)
$$\frac{W \xRightarrow[\Gamma]{\alpha@l} W'}{W \parallel W'' \xRightarrow[\Gamma]{\alpha@l} W' \parallel W''}$$

(W-RPARALLEL)
$$\frac{W \xRightarrow[\Gamma]{\alpha@l} W'}{W'' \parallel W \xRightarrow[\Gamma]{\alpha@l} W'' \parallel W'}$$

(A-NIL)
$$\frac{T_O = \{\} \qquad W \xRightarrow[\Gamma]{\alpha@l} W' \qquad \alpha \in \{\bullet, \sharp[\vec{v}]^{\iota_b,\iota_s}_{\ell'}\}}{(\ell_a, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\alpha@l} (\ell_a, \mathcal{K}) \triangleright W'}$$

(A-BROWSERSERVER)
$$\frac{T_O = \{\} \qquad W \xRightarrow[\Gamma]{\overline{\mathsf{req}}(\iota_b,n,u,p,ck,o)^{l,\mu}@l'} W' \qquad W' \xRightarrow[\Gamma]{\mathsf{req}(\iota_b,n,u,p,ck,o)^{l,\mu}@l} W'' \qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell_a)) \; ? \; (\mathcal{K} \cup \lfloor ns(p,ck) \rfloor_{\ell_a}) : \mathcal{K}}{(\ell_a, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\bullet@l'} (\ell_a, \mathcal{K}') \triangleright W''}$$

(A-SERVERBROWSER)
$$\frac{T_O = \{\} \qquad W \xRightarrow[\Gamma]{\mathsf{res}(\iota_b,n,u,u',\vec{v},ck,page,s)^{l,\mu}@l} W' \qquad W' \xRightarrow[\Gamma]{\overline{\mathsf{res}}(\iota_b,n,u,u',\vec{v},ck,page,s)^{l,\mu}@l'} W'' \qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell_a) \vee \iota_b \neq \mathsf{usr}) \; ? \; (\mathcal{K} \cup \lfloor \{n\} \cup ns(ck,page,s) \rfloor_{\ell_a}) : \mathcal{K}}{(\ell_a, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\bullet@l'} (\ell_a, \mathcal{K}') \triangleright W''}$$

(A-TIMEOUTSEND)
$$\frac{W \xRightarrow[\Gamma]{\overline{\mathsf{req}}(\iota_b,n,u,p,ck,o)^{l,\mu}@l'} W' \qquad W' \xRightarrow[\Gamma]{\mathsf{req}(\iota_b,n,u,p,ck,o)^{l,\mu}@l} \not\qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell_a)) \; ? \; (\mathcal{K} \cup \lfloor \{n\} \cup ns(p,ck) \rfloor_{\ell_a}) : \mathcal{K} \qquad T_O = \{\} \qquad T'_O = \{(\iota_b,n,u,l,\mu)\}}{(\ell_a, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\bullet@l'} (\ell_a, \mathcal{K}') \triangleright W'}$$

(A-TIMEOUTRECV)
$$\frac{T_O = \{(\iota_b,n,u,l,\mu)\} \qquad T'_O = \{\} \qquad W \xRightarrow[\Gamma]{\mathsf{res}(\iota_b,n,u,\bot,\{\},\{\},\{\},\mathbf{skip})^{l,\mu}@l} W'}{(\ell_a, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\bullet@l'} (\ell_a, \mathcal{K}') \triangleright W'}$$

(A-BROATK)
$$\frac{T_O = \{\} \qquad \alpha = \overline{\mathsf{req}}(\iota_b,n,u,p,ck,o)^{\mu,l} \qquad W \xRightarrow[\Gamma]{\alpha@l'} W' \qquad I(\ell_a) \sqsubseteq_I I(\lambda(u)) \qquad \mathcal{K}' = (C(\lambda(u)) \sqsubseteq_C C(\ell_a)) \; ? \; (\mathcal{K} \cup \lfloor ns(p,ck) \rfloor_{\ell_a}) : \mathcal{K}}{(\ell_a, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\alpha@l'} (\ell_a, \mathcal{K}' \cup \{n\}) \triangleright W'}$$

Table B.8: Extended semantics of web systems with the attacker.

(A-ATKSER)
$$\frac{T_O = \{\} \qquad n \leftarrow \mathcal{N} \qquad \iota_b \neq \mathsf{usr} \qquad ns(p, ck) \subseteq \mathcal{K} \qquad \alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{\mathsf{att},\top_I} \qquad W \xRightarrow[\Gamma]{\alpha@\top_I} W'}{(\ell, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\alpha@\top_I} (\ell, \mathcal{K} \cup \{n\}) \triangleright W'}$$

(A-SERATK)
$$\frac{T_O = \{\} \qquad n \in \mathcal{K} \qquad \alpha = \overline{\mathsf{res}}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{\mu,l} \qquad W \xRightarrow[\Gamma]{\alpha@l'} W' \qquad \mathcal{K}' = \mathcal{K} \cup \lfloor ns(ck, page, s, \vec{v}) \rfloor_{\ell_a}}{(\ell, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\alpha@l'} (\ell, \mathcal{K}') \triangleright W'}$$

(A-ATKBRO)
$$\frac{T_O = \{\} \qquad \alpha = \mathsf{res}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{\mathsf{att},\top_I} \qquad W \xRightarrow[\Gamma]{\alpha@l} W' \qquad I(\ell) \sqsubseteq_I I(\lambda(u)) \qquad \{n\} \cup ns(ck, page, s, \vec{v}) \subseteq \mathcal{K} \qquad vars(s) = \emptyset}{(\ell, \mathcal{K}) \triangleright W \xRightarrow[\Gamma]{\alpha@l} (\ell, \mathcal{K}) \triangleright W'}$$

Table B.8: Extended semantics of web systems with the attacker (continued).

- (A-TIMEOUTSEND) (A-TIMEOUTRECV) are two individual rules that together equivalent to the rule (A-TIMEOUT). In rule (A=TIMEOUTSEND) all relevant information is stored in the buffer T_O so that rule (A-TIMEOUTRECV) can send the corresponding response. Note that the integrity label and the sync integrity label may be different.
- (A-BROATK)"forwards" the request with the same annotations.
- (A-ATKSER) sends an event labelled with low integrity and attacker mode att and annotated with low integrity.
- (A-SERATK) "forwards" the request with the same annotations.
- (A-ATKBRO) creates a response with low integrity and attacked mode att. The event can have any sync integrity label – since the browser may expect a different label in different situations.

We show that the original semantics and the extended semantics are equivalent for well typed fresh web systems. Concretely we show that they can produce the same traces. We use here the notation for well-typed websystems $\Gamma \vDash_{\ell_a,\mathsf{usr}} A$, that is formally introduced in Definition 13.

Lemma 1 (Semantic Equivalence)**.** *Let A be a fresh web system with $\Gamma \vDash_{\ell_a,\mathsf{usr}} A$.*

1. *if for some $\vec{\alpha}, A'$ we have $A \xrightarrow{\vec{\alpha}}{}^* A'$ then there exists A'' such that $\overline{A} \xRightarrow[\Gamma]{\vec{\alpha}}{}^* A''$,*

2. *if for some* $\vec{\alpha}$, A' *we have* $\overline{A} \xrightarrow{\vec{\alpha}}{}^* A'$ *then there exists* A'' *such that* $A \overset{\vec{\alpha}}{\underset{\Gamma}{\Rightarrow}}{}^* A''$,

Proof. The claim follows directly by induction over the derivation of $\vec{\alpha}$, using the following observations:

- The integrity label and the attacker state are simply annotations and do not prevent or allow additional steps in the semantics.
- The same is true for the type annotations on values, however we must prevent certain joins on credential types, as they are not defined. Typing ensures that these cases don't occur.
- The command reset l is just modifying the integrity label of the thread, but is otherwise a no-op (S-RESET), so adding it in (T-IFTRUE) and (T-IFFALSE) does not impact the behaviour of the program.
- The split of (A-TIMEOUT) into two separate rules does not impact the semantics as no other rule can be used as long as there is a pending timeout response in the buffer T_O.

□

We also present in Table B.9 enriched typing rules, that allow us to type situations occurring only at runtime and are required to type attacker code. As a convention we use $\vDash$ for the extended typing judgements, while we use $\vdash$ for the original typing judgements. New rules with the same name as an original rule replace that rule, all other original rules also become new rules without modification. Rules with new names are additional rules.

Detailed explanation of enriched to typing rules

- (T-EFRESH) assigns the type ℓ_a to a *fresh*() expression if it is typed in the attacker's run./
- (T-RUNNING) allows us to type running server threads. The typing branch is determined based on the browser identity and the attacked mode of the thread. The typing environment for global variables is determined by the browser identity. If it is the honest users' browser, then the original typing environment is used (since the cookies come from the honest browser). Otherwise, we use an environment where every type is ℓ_a. We then type the code of the thread, inferring the session label $jlabel(j)$ from the session identifier j and using the integrity label as $\mathtt{pc}$.
- (T-EVAL) now gives values their annotated type.
- (T-AUTHATT) does not perform any checks for authenticated events when typing the attackers branch.
- (T-HALT) trivially checks the **halt** and **bad** commands (which only occur at runtime)

- (T-REPLY) now only requires the script to be well typed if we are not typing the attackers branch (i.e., only if the script is sent to the honest user's browser) and additionally passes the URL to the typing judgements for scripts.
- (T-REPLYERR) trivially checks the response with an error message.
- (T-RESET) raises the $\mathtt{pc}$ for the continuation to the label provided in the reset statement.
- (T=BEVAL) now gives values their annotated type.
- (T-BEREFFAIL) allows us to give type any type τ to a browser reference if it may not be read by the script. This rule (and the next one) is needed to ensure that scripts provided by the attacker can be typed (although they will not execute correctly).
- (T-BASSIGNFAIL) allows us to type any assignment to a browser reference, if the script is not allowed to write to it.

We now show that typing with the original typing rules implies typing with the extended rules.

Lemma 2 (Typing Equivalence)**.** *For any fresh server* $S = (\{\}, \{\}, t)$*, whenever we have* $\Gamma, \ell_s, pc \vdash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} t : \ell_s, pc$ *then we also have* $\Gamma, \ell_s, pc \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \overline{t} : \ell_s, pc$*.*

Proof. The proof follows by induction on the typing derivation using the following observations:

- Every typing rule in the original system is also a typing rule in the extended system, with the exception of the modified rules (T-EFRESH), (T-EVAL), (T-REPLY), (T-BEVAL).
- The changes in rules (T-EVAL) and (T-BEVAL) return the type annotations, which are according to the definition of $\bar{\cdot}$, $\bot$ for values $v \notin \mathcal{N}$. Thus the result is the same as in the original typing rule.
- The changes in the rule (T-EFRESH) and (T-REPLY) only affect typing in the typing branch $b = \mathsf{att}$, which does not occur in the original type system. For $b \in \{\mathsf{hon}, \mathsf{csrf}\}$ the rules yield the same result.
- The addition of other rules does not impact the claim

□

(T-EFRESH)
$$\frac{\tau = (b = \mathsf{att})\ ?\ \ell_a\ :\ \mathtt{cred}(\ell)}{\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} \mathit{fresh}()^{\ell} : \tau}$$

(T-RUNNING)
$$\frac{\begin{array}{c}\iota_b \neq \mathsf{usr} \Rightarrow b = \mathsf{att} \qquad \mu = \mathsf{hon} \wedge \iota_b = \mathsf{usr} \Rightarrow b = \mathsf{hon} \qquad \mu = \mathsf{att} \wedge \iota_b = \mathsf{usr} \Rightarrow b = \mathsf{csrf} \\ \Gamma'_{\mathcal{R}^@} = (\iota_b = \mathsf{usr})\ ?\ \Gamma_{\mathcal{R}^@}\ :\ \{\_ \mapsto \ell_a\} \\ (\Gamma_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma'_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}}), \mathit{jlabel}(j), l \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \_, l\end{array}}{\Gamma \vDash^{\mathsf{t}}_{\ell_a,\mathcal{P}} \ell_a \lceil c \rceil^{l,\mu}_{(i,j),(n,u,\iota_b,u)}}$$

(T-EVAL)
$$\frac{v \notin \mathcal{N}}{\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} v^{\tau} : \tau}$$

(T-AUTHATT)
$$\frac{b = \mathsf{att}}{\Gamma, \ell_s, \mathtt{pc} \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell : \ell_s, \mathtt{pc}}$$

(T-HALT)
$$\frac{c \in \{\mathbf{halt}, \mathbf{bad}\}}{\Gamma, \ell_s, \mathtt{pc} \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell_s, \mathtt{pc}}$$

(T-REPLY)
$$\frac{\begin{array}{c}\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r \qquad \mathtt{pc}' = \mathtt{pc} \sqcup_I l_r \qquad \Gamma'_{\mathcal{X}} = x_1 \colon \tau_1, \ldots, x_{|\vec{se}|} \colon \tau_{|\vec{se}|} \\ \Gamma' = (\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}}) \qquad \forall k \in [1 \ldots |\vec{se}|].\, \Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se_k : \tau_k \wedge C(\tau_k) \sqsubseteq_C C(\ell_u) \\ \forall r \in \mathit{dom}(ck).\, \Gamma, \ell_s \vDash^{\mathsf{sr}}_{\ell_a} r : \mathtt{ref}(\tau_r) \wedge \Gamma', \ell_s \vDash^{\mathsf{se}}_{\ell_a} ck(r) : \tau_r \wedge \mathtt{pc}' \sqsubseteq_I I(\tau_r) \\ b \neq \mathsf{att} \Rightarrow \Gamma', b, \mathtt{pc}' \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s \qquad b = \mathsf{csrf} \Rightarrow \forall x \in vars(s).\, C(\Gamma'_{\mathcal{X}}(x)) \sqsubseteq_C C(\ell_a) \\ b = \mathsf{hon} \Rightarrow \mathtt{pc} \sqsubseteq_I l_r \wedge \Big(\mathit{page} = \mathsf{error} \vee \forall v \in \mathit{dom}(\mathit{page}).\, \Gamma', v, \mathtt{pc}' \vDash^{\mathsf{f}}_{\ell_a} \mathit{page}(v)\Big) \\ I(\ell_a) \sqsubseteq_I I(\ell_u) \Rightarrow \forall k \in [1 \ldots |\vec{se}|].\, C(\tau_k) \sqsubseteq_C C(\ell_a)\end{array}}{\Gamma, \ell_s, \mathtt{pc} \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \mathbf{reply}\ (\mathit{page}, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se} : \ell_s, \mathtt{pc}}$$

(T-REPLYERR)
$$\frac{}{\Gamma, \ell_s, \mathtt{pc} \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \mathbf{reply}\ (\mathsf{error}, \mathbf{skip}, \{\}) : \ell_s, \mathtt{pc}}$$

(T-RESET)
$$\frac{}{\Gamma, \ell_s, \mathtt{pc} \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \mathsf{reset}\ l : \ell_s, l}$$

(T-BEVAL)
$$\frac{v \notin \mathcal{N}}{\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} v^{\tau} : \tau}$$

(T-BEREFFAIL)
$$\frac{b = \mathsf{att} \qquad (\lambda(r)) \not\sqsubseteq_C C(\lambda(u))}{\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} r : \tau}$$

(T-BASSIGNFAIL)
$$\frac{b = \mathsf{att} \qquad I(\lambda(u)) \not\sqsubseteq_I I(\lambda(r))}{\Gamma, \mathtt{pc}, b \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P}} r := be}$$

Table B.9: Extended Typing Rules

B.2.4 Subject Reduction

In this section we prove subject reduction for the web system. This is needed to ensure that the system is always in a well-typed state, which in turn is required to prove that our high integrity relation is preserved.

We look at different components of the web system individually. Concretely we will define well-formedness and typing predicates for requests and responses, browsers, servers and websystems as a whole.

We start by defining well-formed requests and responses. Then we define well-typed browsers and show that typing is preserved when the browser takes a step, if the browser only receives well-formed responses, and show that the browser only sends out well-formed requests. We then define well-typed servers and show that typing is preserved whenever the server takes a step, if all requests received by the server are well-formed, and that all requests produced by the server are well-formed. We furthermore show that all requests and responses produced by the attacker are well-formed. Finally, we define well-typed web-systems and show that typing is preserved whenever the websystem takes a step.

Definition 9 (Well-formed Requests)**.** *For a request* $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, o)^{l,\mu}$ *(resp.* $\alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{l,\mu}$*) with* $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ *we have* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *if*

1. *if* $\mu = \mathsf{hon}$ *and* $\iota_b = \mathsf{usr}$ *then*
   - *for all* $k \in dom(p)$ *we have if* $p(k) = v_k^{\tau_k'}$ *then* $\tau_k' \sqsubseteq_{\ell_a} \tau_k$
   - $l \sqsubseteq_I I(\ell_u)$
2. *if* $\mu = \mathsf{att}$ *then*
   - *for all* $k \in dom(p)$ *we have if* $p(k) = v_k^{\tau_k'}$ *then* $\tau_k' \sqsubseteq_{\ell_a} \ell_a$
   - $l \sqsubseteq_{\ell_a} I(\ell_a)$
3. *if* $\iota_b = \mathsf{usr}$
   - *for all* $c \in dom(ck)$ *we have*
     - *if* $ck(c) = v_c^{\tau_c}$ *then* $\tau_c \sqsubseteq_{\ell_a} ref_\tau(\Gamma_{\mathcal{R}^@}(c))$
     - $C(\lambda(r)) \sqsubseteq_{\ell_a} C(\lambda(u))$
4. *if* $\iota_b \neq \mathsf{usr}$ *then for all* $c \in dom(ck)$ *we have if* $ck(c) = v_c^{\tau_c}$ *then* $\tau_c \sqsubseteq_{\ell_a} \ell_a$
5. *If* $\iota_b = \mathsf{usr}$*,* $u \in \mathcal{P}$ *and* $o \neq \bot$ *and* $I(\ell_a) \not\sqsubseteq_I o$ *then* $\mu = \mathsf{hon}$.

Intuitively, according to Definition 9 a request is well-formed, if

1. for all honest requests, all parameter types are respected and the integrity label is higher than the integrity label of the URL.

2. for all attacked requests, all parameters are of the attacker's type and the integrity is low.
3. For all (attacked and honest) requests from the users browser, all cookies respect their type from the environment and their confidentiality is as most as high as the one of the URL.
4. For all requests by the attacker, all cookies have the type of the attacker.
5. Any request with a high integrity origin to a protected URL must be honest.

We now in a similar fashion define well-formed responses.

Definition 10 (Well-formed Responses)**.** *For a response* $\alpha = \overline{\mathsf{res}}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l,\mu}$ *(resp.* $\alpha = \mathsf{res}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l,\mu}$*) with* $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ *we have* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *if*

1. *For all* $v^\tau \in values(ck, page, s, \vec{v})$ *we have* $C(\tau) \sqsubseteq_C C(\ell_u)$
2. *If* $\iota_b \neq \mathsf{usr}$*, then for all* $v^\tau \in values(ck, page, s, \vec{v})$ *we have* $\tau \sqsubseteq_{\ell_a} \ell_a$
3. *if* $\iota_b = \mathsf{usr}$ *and* $\mu = \mathsf{att}$ *then for all* u'' *with* $I(\ell_a) \sqsubseteq_I I(\lambda(u')))$ *we have* $\Gamma, \mathsf{att}, \top_I \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u''} s$ *and* $I(\ell_a) \sqsubseteq_I l$
4. *if* $\iota_b = \mathsf{usr}$ *then for all* $r \in dom(ck)$ *with* $ck(r) = v^\tau$ *we have*
   - *If* $\lambda(u) \sqsubseteq_I \lambda(r)$ *then* $\tau \sqsubseteq_{\ell_a} ref_\tau(\Gamma_{\mathcal{R}^@}(r))$ *and* $l \sqsubseteq_I I(ref_\tau(\Gamma_{\mathcal{R}^@}(r)))$
   - *If* $\lambda(u) \not\sqsubseteq_I \lambda(r)$ *then* $\tau \sqsubseteq_{\ell_a} ref_\tau(\Gamma_{\mathcal{R}^@}(r))$ *or* $\tau \sqsubseteq_{\ell_a} \ell_a$
5. *if* $\iota_b = \mathsf{usr}$ *and* $I(\ell_a) \not\sqsubseteq_I l$ *then* $page = \mathsf{error}$ *or for all* $v \in dom(page)$ *with* $page(v) = \mathsf{form}(u'', \vec{v^\tau})$ *we have*
   - $\Gamma_{\mathcal{U}}(u'') = \Gamma_{\mathcal{V}}(v)$
   - *with* $\Gamma_{\mathcal{U}}(u'') = \ell'_u, \vec{\tau'}, l'_r$,
     - *for all* $i \in [1 \ldots |\vec{v}|]$ *we have* $\tau_i \sqsubseteq_{\ell_a} \tau'_i$
     - $l \sqsubseteq_I \ell'_u$
6. *if* $\iota_b = \mathsf{usr}$ *and* $I(\ell_a) \sqsubseteq_I l$ *then* $page = \mathsf{error}$ *or we have one of the following*
   - *for all* $v \in dom(page)$ *with* $page(v) = \mathsf{form}(u'', \vec{v})$*, for all* $i \in [1 \ldots |\vec{v}|]$ *we have* $\tau_i \sqsubseteq_{\ell_a} \ell_a$
   - *or* $I(\ell_a) \not\sqsubseteq_I u$
7. $\iota_b = \mathsf{usr}$ *and* $\mu = \mathsf{hon}$ *then* $\Gamma, \mathsf{hon}, l_r \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s$ *and* $l = l_r$
8. *if* $u' \neq \bot$ *and* $\iota_b = \mathsf{usr}$ *then with* $\alpha' = \overline{\mathsf{req}}(\iota_b, n', u', p, \{\}, \bot)^{l,\mu}$*, for any* n' *and* $\forall k \in [1 \ldots |\vec{v}|] : p(k) = v_k$ *we have* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha'$.

Intuitively, according to Definition 10 a response is well-formed if

1. The confidentiality label of all values contained in the response is at most as high as the confidentiality label of the URL from which the response is sent, or the confidentiality is low.

2. If the response it not sent to the honest user, then all values must be of low confidentiality.

3. If the response is sent to the honest user and influenced by the attacker, then the integrity label is low and the contained script is well-typed, using the type branch att.

4. For all responses to the honest users, if a cookie may be set by the response, then it respects the typing environment (also taking into account the integrity label of the response). If the cookie may not be set by the response, then it respects the typing environment or is low.

5. For all honest responses we have that the page is either the error page, or that it is well-typed, i.e., the type of the form name matches the type of the URL and all parameters respect the URL type and that the integrity of the current thread is high enough to trigger a request to that URL.

6. For all attacked responses to the honest user, we have that the page is the error page or one of the following holds :

   - all parameters contained in the DOM are of type ℓ_a
   - or the response comes from a high integrity URL (in which case we do not make any assumption on the DOM, since the user will not interact with it)

7. For all honest responses, the script is well-typed with pc set to the expected response integrity of the URL, and the integrity of the response must be equal to that label

8. If the redirect URL is not empty, and the response is sent to the honest user's browser, then we know that the request that will result from processing the response at the browser is well-typed (using an empty set of cookies and an empty origin as placeholders).

Definition 11 (Browser Typing)**.** *Let* $B = (N, M, P, T, Q, \vec{a})^{\iota_b, l, \mu}$ *be a browser. We write* $\Gamma \vDash_{\ell_a, \mathsf{usr}} B$*, if* $\iota_b = \mathsf{usr}$ *and*

1. $\mu = \mathsf{att} \Rightarrow I(\ell_a) \sqsubseteq_I l$
2. $\forall r \in dom(M), M(r) = v^\tau \wedge \tau = ref_\tau(\Gamma_{\mathcal{R}^@}(r))$
3. *For all* $tab \in dom(P)$ *with* $P(tab) = (u, page, l', \mu')$ *and* $page \neq \mathsf{error}$ *we have for all* $v \in dom(page)$ *with* $page(v) = \mathsf{form}(u'^{\tau_{u'}}, \vec{v^\tau})$
   - $C(\tau_{u'}) \sqsubseteq_C C(\ell_a)$
   - *if* $I(\ell_a) \not\sqsubseteq_I l'$ *and* $\Gamma_{\mathcal{U}}(u') = \ell_u, \vec{\tau'}, l_r$ *then*
     - $\mu = \mathsf{hon}$
     - $\Gamma_{\mathcal{U}}(u') = \Gamma_{\mathcal{X}}(v)$

  - $l' \sqsubseteq_I I(\ell_u)$
  - $\forall i \in [1 \ldots |\vec{v}|].\, \tau_i \sqsubseteq_{\ell_a} \tau'_i$
- *if* $I(\ell_a) \sqsubseteq_I l'$ *then one of the following holds*
  - $\forall i \in [1 \ldots |\vec{v}|].C(\tau_i) \sqsubseteq_C C(\ell_a)$
  - $I(\ell_a) \not\sqsubseteq_I I(\lambda(u))$

4. *If* $T = \{tab \mapsto s\}$ *and* $P(tab) = (u, page, l', \mu')$ *with* $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ *then*
   - $l = l'$
   - *if* $\mu = \mathsf{hon}$ *then* $l' \sqcup_I l \sqsubseteq_I l_r$ *and* $\Gamma, \mathsf{hon}, l_r \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s$
   - *if* $\mu = \mathsf{att}$ *then* $\Gamma, \mathsf{att}, \top_I \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s$
5. *If* $Q = \{\alpha @ l'\}$, *then we have* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$.
6. *For* $\vec{a}$ *we have*
   - *for the navigation flow*
     - *for every navigation flow* $\vec{a}'$ *in* $\vec{a}$, *we have that* $I(\ell_a) \sqsubseteq_I I(\lambda(a'_j))$ *implies* $I(\ell_a) \sqsubseteq_I I(\lambda(a'_k))$ *for all* $j < k \leq |\vec{a'}|$.
     - *If* $N = \{n \mapsto (tab, u, o)\}$ *and* $T = \{\}$ *then we have that for all* $a \in nf(\vec{a}, tab)$ *that* $I(\ell_a) \sqsubseteq_I l$ *implies* $I(\ell_a) \sqsubseteq_I I(\lambda(a))$. *Furthermore we have* $I(\ell_a) \sqsubseteq_I I(\lambda(a'_j))$ *implies* $I(\ell_a) \sqsubseteq_I I(\lambda(a'_k))$ *for all* $j < k \leq |\vec{a'}|$.
     - *for all* $tab \in dom(P)$ *with* $P(tab) = (u, page, l', \mu')$ *and* $N \neq \{n_N \mapsto (tab, u_N, o_N)\}$ *for all* n_N, u_N, o_N, *we have that for all* $a \in nf(\vec{a}, tab)$ *that* $(I(\ell_a) \sqsubseteq_I I(\lambda(u))$ *or* $\mu' = \mathsf{att})$ *implies* $I(\ell_a) \sqsubseteq_I I(\lambda(a))$. *Furthermore we have* $I(\ell_a) \sqsubseteq_I I(\lambda(a'_j))$ *implies* $I(\ell_a) \sqsubseteq_I I(\lambda(a'_k))$ *for all* $j < k \leq |\vec{a'}|$.
   - *for all actions* a' *in* $\vec{a}$ *we have:*
     - *if* $a' = \mathsf{load}(tab, u, p)$ *and* $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ *then for all* $k \in dom(p)$ *we have that if* $p(k) = v^{\tau'}$ *then* $\tau' \sqsubseteq_{\ell_a} \tau_k$;
     - *if* $a' = \mathsf{submit}(tab, u, v', p)$ *and* $\Gamma_{\mathcal{V}}(v') = \ell_u, \vec{\tau}, l_r$ *then for all* $k \in dom(p)$ *we have that if* $p(k) = v^{\tau'}$ *then*
       * $\tau' \sqsubseteq_{\ell_a} \tau_k$.
       * *if* $I(\ell_a) \sqsubseteq_I \lambda(u)$ *then additionally* $C(\tau') \sqsubseteq_C C(\ell_a)$
7. *If* $N = \{n \mapsto (tab, u, o)\}$ *and* $T = \{tab \mapsto s\}$ *and* $\mu = \mathsf{hon}$ *then*
   - *if* $P(tab) = (u', page, l', \mu')$ *and* $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ *and* $\Gamma_{\mathcal{U}}(u') = \ell'_u, \vec{\tau'}, l'_r$ *then* $l_r = l'_r$.
   - $I(\ell_a) \not\sqsubseteq_I I(\lambda(u))$

Intuitively, according to Definition 11 a browser is well-typed, if all its components are well-typed. Concretely, we require that:

1. Whenever the state of the browser is directly influenced by the attacker, then the integrity of the browser is low.

2. All values stored in a memory reference have a type annotation that is equal to the type of the reference the typing environment.

3. For any non empty DOM in a tab,

   - If the DOM is of high integrity
     - The DOM is honest
     - The type of the form name matches the type of the URL
     - The integrity label of the DOM is higher than the integrity label of the URL
     - All parameters have the expected type.
   - If the DOM is low integrity
     - and either
       * All parameters have the attacker's type.
       * or the integrity of the DOM's origin is high

4. If a script is running in a tab

   - the script integrity is equal to the integrity of the DOM in that tab.
   - if the browser is not attacked then the browser's integrity is equal the integrity of the expected response type for the URL of the DOM in the same tab and the script code is well-typed in the honest typing branch using the expected response type as the $\mathtt{pc}$.
   - if the browser is attacked, then the script is well-typed using $\top_I$ label as $\mathtt{pc}$.

5. All requests in the buffer are well-formed.

6. For all user actions we have that

   - The user will not submit forms on high integrity pages after "tainting" the connection, by visiting a low integrity page. Concretely the conditions are the following:
     - The first condition is exactly the assumption we make on well-formed user actions.
     - The second condition is the same, but taking into account open network connections for load or submits.
     - The third condition is similarly taking into account pages already loaded in browser tabs for the navigation flow. However it is less strict, as it uses the attacked state of the page instead of the integrity labels of previously visited pages. Concretely, this would allow navigation of high integrity pages even after visiting low integrity pages, as long as there has not been a direct influence by the attacker.

- The user's inputs respect the parameter types and will only input low confidentiality values in forms present in low integrity pages..

7. Whenever the browser is in an honest state, has a script running in the context of URL u and is waiting for the response of a script inclusion from URL u', then

   - the two URLs have the same expected response type.
   - the URL u' is of high integrity.

We now prove that whenever a browser expression containing variables is well typed in a typing environment, then it is also well-typed if we substitute the variables with concrete values of the expected type.

Lemma 3 (Browser Expression Substitution)**.** *Whenever we have* $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be : \tau$ *and we have a substitution* σ *with* $dom(\sigma) = dom(\Gamma_{\mathcal{X}})$ *and* $\forall x \in dom(\Gamma_{\mathcal{X}}).\, \sigma(x) = v_x^{\tau_x}$ *with* $\tau_x \sqsubseteq_{\ell_a} \Gamma_{\mathcal{X}}(x)$ *then for all* $\Gamma'_{\mathcal{X}}$*, we have* $(\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}}), b \vDash^{\mathsf{be}}_{\ell_a} be\sigma : \tau$*,*

Proof. We perform an induction on the typing derivation of $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be : \tau$:

- (T-BEVAR). Then $be = x$ and $be\sigma = v_x^{\tau_x}$ with $\tau_x \sqsubseteq_{\ell_a} \Gamma_{\mathcal{X}}(x)$. The claim follows directly from rule (T-BVAL) and (T-BSUB).
- (T-BEREF). Then $be = r = be\sigma$ and the claim is trivial.
- (T-BEVAL). Then $be = v^{\tau_v} = be\sigma$ and the claim is trivial.
- (T-BEUNDEF). Then $be = \bot = be\sigma$ and the claim is trivial.
- (T-BENAME). Then $be = n^{\tau_n} = be\sigma$ and the claim is trivial.
- (T-BEDOM). Then $be = \mathsf{dom}(be_1, be_2)$ and $be\sigma = \mathsf{dom}(be_1\sigma, be_2\sigma)$ and the claim follows immediately using (T-BEDOM).
- (T-BEBINOP) Then $be = be_1 \odot be_2$ with $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be_1 : \tau_1$ and $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be_2 : \tau_2$ and $\tau = label(\tau_1) \sqcup label(\tau_2)$. We also have $be\sigma = be_1\sigma \odot be_2\sigma$. By induction we know that $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be_1\sigma : \tau'_1$ and $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be_2\sigma : \tau'_2$ with $\tau'_1 \sqsubseteq_{\ell_a} \tau_1$ and $\tau'_2 \sqsubseteq_{\ell_a} \tau_2$. We then know that $label(\tau'_1) \sqcup label(\tau'_2) \sqsubseteq_{\ell_a} label(\tau_1) \sqcup label(\tau_2) = \tau$, and the claim follows by (T-BINOP) and (T-BESUB).
- (T-BESUB) follows by induction and by the transitivity of $\sqsubseteq_{\ell_a}$.

□

Next, we prove the same claim on the level of scripts.

Lemma 4 (Browser Substitution)**.** *Whenever we have* $\Gamma, \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a,\mathcal{P}} s$ *and we have a substitution* σ *with* $dom(\sigma) = dom(\Gamma_{\mathcal{X}})$ *and* $\forall x \in dom(\Gamma_{\mathcal{X}}).\, \sigma(x) = v_x^{\tau_x}$ *with* $\tau_x \sqsubseteq_{\ell_a} \Gamma_{\mathcal{X}}(x)$ *then for all* $\Gamma'_{\mathcal{X}}$*, we have* $(\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}), \mathtt{pc}, b \vdash^{\mathsf{s}}_{\ell_a,\mathcal{P}} s\sigma$.

Proof. We do the proof by induction on the typing derivation.

- (T-BSeq): Then $s = s_1, s_2$. The claim follows by applying the induction hypothesis to s_1 and s_2 and applying rule (T-BSeq).
- (T-BSkip): The claim follows trivially.
- (T-BAssign): Then we have $s = r := be$, with
  - $\Gamma \vDash^{\mathsf{br}}_{\ell_a} r : \mathtt{ref}(\tau)$
  - $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be : \tau$
  - $\mathtt{pc} \sqsubseteq_I I(\tau)$

  Using Lemma 3 and (T-BESub), we get $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be\sigma : \tau$ and the claim follows immediately.
- (T-BSetDom): Then $s = \mathbf{setdom}(v, u, \vec{be})$. The claim follows by applying of Lemma 3 and (T-BESub) for every be_i in $\vec{be}$.
- (T-BInclude): Then $s = \mathbf{include}(u, \vec{be})$, The claim follows by applying of Lemma 3 and (T-BESub) for every be_i in $\vec{be}$.

□

Now, we show that typing is preserved under the evaluation of expressions.

Lemma 5 (Browser Expression Typing)**.** *Let* $B = (N, M, P, T, Q, \vec{a})^{\iota_b, l, \mu}$ *be a browser with* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$*. Let* $T = \{tab \mapsto s\}$ *and* $\{tab \mapsto (u, f, l', \mu')\} \in P$*,* $\ell = \lambda(u)$*. Then for any browser expression be, if* $\Gamma, \mu \vDash^{\mathsf{be}}_{\ell_a} be : \tau$ *then* $\Gamma, \mu \vDash^{\mathsf{be}}_{\ell_a} eval_\ell(be, M, f) : \tau$

Proof. Let $eval_\ell(be, M, f) = v^{\tau'}$. We show $\tau' \sqsubseteq_{\ell_a} \tau$ and the claim follows using rule (T-BESub). We perform the proof by induction over the expression be:

- $be = x$: In this case, $eval_\ell(se, M, f)$ is undefined, so we don not have to show anything.
- $be = v^\tau$ We have $eval_\ell(v^\tau, M, f) = v^\tau$ and the claim is trivial.
- $be = be_1 \odot be_2$: By induction we know
  - $\Gamma, \mu \vDash^{\mathsf{be}}_{\ell_a} be_1 : \tau_1$ and $eval_\ell(be_1, M, f) = v_1^{\tau'_1}$ and $\tau'_1 \sqsubseteq_{\ell_a} \tau_1$
  - $\Gamma, \mu \vDash^{\mathsf{be}}_{\ell_a} be_2 : \tau_2$ and $eval_\ell(be_2, M, f) = v_2^{\tau'_2}$ and $\tau'_2 \sqsubseteq_{\ell_a} \tau_2$

Let now $v^{\tau'} = v_1^{\tau_1'} \odot v_2^{\tau_2'}$. Then we know that $\tau' = label(\tau_1') \sqcup label(\tau_2')$ by rule (BE-BINOP). By rule (T-BEBINOP) we have $\tau = label(\tau_1) \sqcup label(\tau_2)$, and the claim follows.

- $be = r$: then the claim immediately follows from rule (T-BEREF) and property 2 of Definition 11.
- $be = \mathsf{dom}(be_1, be_2)$: We know by property 4 of Definition 11 $l = l'$ We distinguish two cases:
  - If $I(\ell_a) \not\sqsubseteq_I l'$ then we know that $\mu = \mathsf{hon}$ and hence this case is impossible, since we do not have a typing rule for the expression in the honest type branch.
  - If $I(\ell_a) \sqsubseteq_I l'$, then we distinguish two cases:
    - if $I(\ell_a) \not\sqsubseteq_I I(\lambda(u))$ then we know that the script can also be typed with $b = \mathsf{hon}$, and hence this case is impossible.
    - if $I(\ell_a) \sqsubseteq_I I(\lambda(U))$ then by rule (BE-DOM) the value is either a URL parameter or the URL itself. we then know that for all parameters $v^{\tau'}$ of any URL in the DOM we have $C(\tau') \sqsubseteq_C C(\ell_a)$. For any URL u^{τ_u} we have $C(\tau_u) \sqsubseteq_C C(\ell_a)$ and the claim holds.

□

We now show subject reduction for the browser for internal steps i.e., whenever a well-typed browser takes a step, it results in another well-typed browser. We treat browsers sending requests and receiving responses in separate lemmas.

Lemma 6 (Browser Subject Reduction)**.** *Let B, B' be browsers with $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$ such that $B \xrightarrow{\bullet@} B'$. Then we have $\Gamma \vDash_{\ell_a,\mathsf{usr}} B'$.*

Proof. Let $B = (N, M, P, T, Q, \vec{a})^{\iota_b,\mu,l}$ and $B' = (N', M', P', T', Q', \vec{a'})^{\iota_b,\mu',l'}$ be browsers as in the lemma. We know that $\iota_b = \mathsf{usr}$ and do a proof by induction on the step taken. We show that all properties of Definition 11 hold for B'.

- (B-LOAD):
  - Property 1 is trivial, since $\mu' = \mathsf{hon}$.
  - Property 2 is trivial, since $M = M'$
  - Property 3 is trivial, since $P = P'$
  - Property 4 is trivial, since $T = \{\}$.
  - For property 5 we have $Q' = \{\alpha\}$ with $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, \bot)^{I(\lambda(u)),\mathsf{hon}}$ and hence have to show that $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$. We show that all the properties of Definition 9 are fulfilled.
    - Property 1 follows immediately from property 6 of Definition 11 for B

  - Property 2 is trivial since we have $\mu = \mathsf{hon}$
  - Property 3 follows immediately from property 2 of Definition 11 for B and the definition of $get\_ck(\cdot,\cdot)$.
  - Property 4 is trivial since $\iota_b = \mathsf{usr}$
  - Property 5 is trivial since the origin $o = \bot$.
- Property 6 for B' follows directly from property 6 of Definition 11 for B. The navigation flow started by the load action is the same as $nf(\vec{a}, tab)$
- Property 7 is trivial since $T' = \{\}$

- (B-INCLUDE)
  - Property 1 is trivial, since $\mu' = \mu$ and $l = l'$
  - Property 2 is trivial, since $M = M'$
  - Property 3 is trivial, since $P = P'$
  - Property 4 is trivial using rule (T-BSKIP), since $T = \{tab \mapsto \mathbf{skip}\}$
  - For property 5 we have $Q' = \{\alpha\}$ with $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, \mathrm{orig}(u'))^{l \sqcup_I I(\lambda(u)),\mu'}$ and hence have to show that $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$. We show that all the properties of Definition 9 are fulfilled.
    - For property 1 We distinguish two cases:
      1. if $\mu = \mathsf{hon}$ then it follows from property 4 of Definition 11 for B using rule (T-BINCLUDE) and Lemma 5
      2. if $\mu = \mathsf{att}$ then the claim is trivial
    - For property 2 We distinguish two cases:
      1. if $\mu = \mathsf{hon}$ then the claim is trivial
      2. if $\mu = \mathsf{att}$ then it follows from property 4 of Definition 11 for B using rule (T-BINCLUDE) and Lemma 5
    - Property 3 follows immediately from property 2 of Definition 11 for B
    - Property 4 is trivial since $\iota_b = \mathsf{usr}$
    - For property 5 we perform a case distinction:
      - If $u \notin \mathcal{P}$, $I(\ell_a) \sqsubseteq_I \mathrm{orig}(u')$ or $\mu' = \mathsf{hon}$ then the claim is trivial.
      - If $u \in \mathcal{P}$, $I(\ell_a) \not\sqsubseteq_I \mathrm{orig}(u')$ and $\mu' = \mathsf{att}$ then assume that the include statement is contained in the script $s_{u'}$ served by u'. Since u' is of high integrity, we know that the script code can be typed with $b = \mathsf{hon}$. This in particular implies that every include statement in the script also has been typed with $b = \mathsf{hon}$. Hence we know by rule (T-BINCLUDE) that $u \notin \mathcal{P}$ and we have a contradiction. If the include statement is not contained in the script $s_{u'}$ served by u', then it must be contained in the script $s_{u''}$ served from some URL u'' that is included by the script $s_{u'}$. Using the same argumentation, we know by rule (T-BINCLUDE) that $I(\ell_a) \not\sqsubseteq_I I(\lambda(u''))$ and again using the same argumentation we get the contradiction $u \notin \mathcal{P}$

  - Property 6 of Definition 11 for B' follows directly from property 6 for B.
  - Property 7 follows from property 4 of Definition 11 for B using rule (T-BINCLUDE)

- (B-SUBMIT) Then we have
  - $a = \mathsf{submit}(tab, u, v, p')$
  - $\{tab \mapsto (u, f, l', \mu')\} \in P$
  - $\{v \mapsto \mathsf{form}(u', \vec{v^\tau})\} \in f$
  - $\forall k \in [1 \ldots |\vec{v}|].\, p(k) = k \in \mathit{dom}(p')\ ?\ p'(k) : v_k^{\tau_k}$
  - Property 1 follows from property 3 of Definition 11 for B.
  - Property 2 is trivial, since $M = M'$
  - Property 3 is trivial, since $P = P'$
  - Property 4 is trivial, since $T = \{\}$
  - For property 5 we have $Q' = \{\alpha\}$ with $\alpha = \overline{\mathsf{req}}(\iota_b, n, u', p, ck, \mathrm{orig}(u))^{l' \sqcup_I I(\lambda(u')), \mu'}$ and hence have to show that $\Gamma \vDash_{\ell_a, \mathsf{usr}} \alpha$. We show that all the properties of Definition 9 are fulfilled.
    * For property 1 we distinguish two cases:
      1. if $\mu' = \mathsf{hon}$ then it follows from property 3 and 6 of Definition 11 for B
      2. if $\mu' = \mathsf{att}$ then we distinguish two cases:
         · if $I(\ell_a) \sqsubseteq_I I(\lambda(u))$ then the claim is trivial
         · otherwise, we know from property 6 that $I(\ell_a) \sqsubseteq_I \lambda(a)$. By the definition of λ we get $\lambda(a) = \lambda(u)$ which is a contradiction to our assumption. Hence this case cannot happen.
    * For property 2 we distinguish two cases:
      1. if $\mu' = \mathsf{hon}$ the claim is trivial
      2. if $\mu' = \mathsf{att}$ then it follows from property 3 and 6 of Definition 11 for B
    * Property 3 follows immediately from property 2 of Definition 11 for B and Lemma 5
    * Property 4 is trivial since $\iota_u = \mathsf{usr}$
    * For property 5 we perform a case distinction:
      · If $u' \notin \mathcal{P}$, $I(\ell_a) \sqsubseteq_I \mathrm{orig}(u)$ or $\mu' = \mathsf{hon}$ then the claim is trivial.
      · If $u' \in \mathcal{P}$, $I(\ell_a) \not\sqsubseteq_I \mathrm{orig}(u)$ and $\mu' = \mathsf{att}$ then we know $I(\ell_a) \not\sqsubseteq_I \lambda(a)$. We then get by property 6 of Definition 11 for B that $I(\ell_a) \sqsubseteq_I \mathrm{orig}(u)$ or $\mu = \mathsf{hon}$ and immediately have a contradiction.
  - Property 6 of Definition 11 for B' follows from property 6 for B, since request from low integrity pages, are also of low integrity and since high integrity pages do not include low integrity pages (by (T-FORM).
  - Property 7 is trivial since $T = \{\}$.

- (B-SEQ) Then $T = \{tab \mapsto s\}$ with $s = s_1; s_2$ and from (B-BSEQ) we know $\Gamma, \mu, l_r \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s_2$. We apply the induction hypothesis for the browser stepping from script s_1 to s_1'. This immediately gives us all properties from Definition 11 except the typing of the script $\Gamma, \mu, l_r \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s_1'; s_2$, but this claim follows immediately by applying rule (T-BSEQ).

- (B-SKIP) Then $T = \{tab \mapsto s\}$ with $s = \mathbf{skip}; s'$ By rule (T-BSEQ) we have $\Gamma, \mu, l_r \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s'$. Since nothing besides the script changes, the claim follows immediately.

- (B-END)

  - Property 1 is trivial since $\mu' = \mathsf{hon}$
  - Property 2 is trivial since $M = M'$
  - Property 3 is trivial since $P = P'$
  - Property 4 is trivial since $T = \{\}$
  - Property 5 is trivial since $Q = \{\}$
  - Property 6 is trivial since $\vec{a} = \vec{a'}$, $P = P'$ and $N = N'$
  - Property 7 is trivial since $M = \{\}$.

  is trivial, since the only change from B to B' is that $T' = \{\}$, in which case we don't have to show anything for the script.

- (B-SETREFERENCE) Then $T = \{tab \mapsto s\}$ with $s = r := be$. We have $P = P'$ and claim 3 of Definition 11 is trivial and since $T' = \{tab \mapsto \mathbf{skip}\}$ claim 4 follows immediately from rule (T-BSKIP).

  By rule (B-SETREFERENCE) we have

  - $\{tab \mapsto (u, f, l', \mu')\} \in P$
  - $\ell = \lambda(u)$
  - $eval_\ell(be, M, f, l') = v^\tau$
  - $M' = M\{r \mapsto v^{\tau_r}\}$ with $\tau_r = \tau \sqcup \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r)) \tilde{\sqcup}_I l$

  All properties of Definition 11 except for property 2 are trivial.

  For property 2 it is sufficient to show that $\tau_r \sqsubseteq_{\ell_a} \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$.

  By rule (T-BASSIGN) and rule (T-BREF) we get that

  - $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be : \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$
  - $l \sqsubseteq_I I(\mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r)))$

  By Lemma 5 we get $\tau \sqsubseteq_{\ell_a} \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$. We hence get $\tau_r = \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$ and the claim follows.

- (B-SETDOM) Then $T = \{tab \mapsto s\}$ with $s = \mathbf{setdom}(be, u, \vec{be})$.

  All properties of Definition 11 except for property 3 are trivial, so we only show this one.

  We assume the following setting analog to rule (B-SETDOM)

  - $P = P_0 \uplus \{tab \mapsto (u', f, l'', \mu'')\}$.
  - $\ell = \lambda(u')$
  - $eval_\ell(be', M, f) = v'$
  - $\forall k \in [1 \dots |\vec{be}|].\, v_k'^{\tau_k'} = eval_\ell(be_k, M, f) \wedge v_k^{\tau} = v_k'^{\tau' \sqcup_I l}$
  - $\mu''' = (\mu = \mathsf{att} \vee \mu'' = \mathsf{att})\ ?\ \mathsf{att}\ :\ \mathsf{hon}$

  Then $P' = P_0 \uplus \{tab \mapsto (u', f\{v' \mapsto \mathsf{form}(u^{(\bot_C, l)}, \vec{v^\tau})\}, l'' \sqcup_I l, \mu''')\}$. We now do a case analysis:

  - $I(\ell_a) \not\sqsubseteq_I l''$: Then by property 4 of Definition 11 we know $l = l''$ and hence $l \sqcup_I l' = l''$ We now need to show that with $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau_u}, l_r$
    1. $\Gamma_{\mathcal{U}}(u) = \Gamma_{\mathcal{X}}(v)$
    2. $l \sqcup_I l'' \sqsubseteq_I I(\ell_u)$ and
    3. $\forall i \in [1 \dots |\vec{v}|].\, \tau_i \sqsubseteq_{\ell_a} \tau_{ui}$
    4. $\mu''' = \mathsf{hon}$

    (1) follows immediately from rule (T-BSETDOM),

    From Definition 11, we know by property 3 that $l'' \sqsubseteq_I I(\ell_u)$ and by property 4 we know with $\Gamma_{\mathcal{U}}(u') = \ell_u', \vec{\tau_u'}, l_r'$ that $l'' = l_r$ and by rule (T-BSETDOM) we know that $l_r' \sqsubseteq_I I(\ell_u)$ and (2) follows.

    For (3), we get with rule (T-BSETDOM) and Lemma 5 that $\forall i \in [1 \dots |\vec{v}|].\, \tau_i = \tau_{ui}$ and the claim follows immediately.

    (4) is trivial, since with $I(\ell_a) \not\sqsubseteq_I l$ and $I(\ell_a) \not\sqsubseteq_I l''$ we also know $\mu = \mathsf{hon}$ and $\mu'' = \mathsf{hon}$.
  - $I(\ell_a) \sqsubseteq_I l''$: Then we need to show that on of the following holds
    - $\forall i \in [1 \dots |\vec{v}|].\, \tau_i \sqsubseteq_{\ell_a} \ell_a$
    - or $I(\ell_a) \not\sqsubseteq_I u'$

    If $I(\ell_a) \not\sqsubseteq_I u'$, the claim is trivial, we hence assume $I(\ell_a) \sqsubseteq_I u'$. The claim then follows immediately from the observation that by rule (T-REPLY) scripts of low integrity URLs can never contain any values of high confidentiality

□

We now show that Browsers remain well-typed if they send out and request and that every sent request is well-formed.

Lemma 7 (Browser Request)**.** *Whenever a browser* $B \xrightarrow{\alpha} B'$ *with* $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, o)^{l,\mu}$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$*. Then* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B'$

Proof. Let $B = (N, M, P, T, Q, \vec{a})^{\iota_b,l,\mu}$ and We know that rule (B-FLUSH) is used. We hence have $Q = \{\alpha @ l'\}$ and $B' = (N, M, P, T, \{\}, \vec{a'})^{\iota_b,l,\mu}$. $\Gamma \vDash_{\ell_a,\mathsf{usr}} B'$ then follows immediately from $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$ We get $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ by property 5 of Definition 11. □

The next lemma states that a well-typed browser receiving a well-formed response is still a well-typed browser. We have the additional assumptions that the integrity of the response is at most as high as the integrity of the browser and that either the attacked mode of the browser and the response are the same or that the response is attacked and the integrity of the responding URL is low.

Lemma 8 (Browser Response)**.** *Whenever a for a browser* $B = (N, M, P, T, Q, \vec{a})^{\iota_b,l,\mu}$ *we have* $B \xrightarrow{\alpha} B'$ *with* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$*,* $\alpha = \mathsf{res}(\iota_b, n, u, u', ck, \vec{v}, page, s)^{l'',\mu''}$ *with* $l \sqsubseteq_I l''$ *and* $\mu = \mu'' \vee \mu'' = \mathsf{att} \wedge I(\ell_a) \sqsubseteq_I I(\lambda(u))$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *then* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B'$*.*

Proof. $B' = (N', M', P', T', Q', \vec{a'})^{\iota_b,l',\mu'}$ We show that B' fulfills the properties of Definition 11. We know that the step α was taken using rule (B-RECVLOAD) (B-RECVINCLUDE), or (B-REDIRECT). In all cases property 2 of Definition 11 follows immediately from property 4 of Definition 10. We now do a case distinction on the rule used

- (B-RECVLOAD)
  - Property 1 follows immediately from property 3 of Definition 10
  - For property 3 we do a case distinction:
    - if $\mu' = \mu'' = \mathsf{hon}$ then the claim follows from property 5 of Definition 10.
    - if $\mu' = \mu'' = \mathsf{att}$ then the claim follows from properties 6 and 3 of Definition 10
  - Property 4 follows from property 7 of Definition 10 for $\mu' = \mathsf{hon}$ and from 3 of Definition 10 if $\mu' = \mathsf{att}$.
  - Property 5 is trivial.
  - Property 6 follows from the same property for B. the nf on the tab for the page in B' is the same as the one for the network connection in B
  - Property 7 is trivial.
- (B-RECVINCLUDE)
  - Property 1 follows immediately from property 3 of Definition 10 and property 1 of Definition 11 for B.
  - Property 3 is trivial
  - For property 4 we do a case distinction:

  - If $\mu'' = \mathsf{hon}$, then $\mu = \mu' = \mathsf{hon}$ and the claim follows from property 7 of Definition 10 and property 4 of Definition 11 for B, using rule (T-BSEQ) and property 7 of Definition 11
  - if $\mu'' = \mathsf{att}$ then $\mu' = \mathsf{att} \vee \mu = \mathsf{att}$. Since we know that $\mu = \mathsf{att} \Rightarrow \mu' = \mathsf{att}$ we can conclude that $\mu' = \mathsf{att}$ We distinguish two cases:
    - If $\mu = \mathsf{att}$ the claim follows immediately using property 3 of Definition 10 and property 4 of Definition 11 for B, using rule (T-BSEQ)
    - f $\mu = \mathsf{hon}$ then by the assumption in the lemma we have $I(\ell_a) \sqsubseteq_I I(\lambda(u))$ which is in contradiction to property 7 of Definition 11, hence this case is impossible.

- Property 5 is trivial.
- Property 6 is trivial
- Property 7 is trivial.

- (B-REDIR)
  - Property 1 is trivial.
  - Property 3 is trivial.
  - Property 4 is trivial.
  - For property 5 we know that $Q' = \{\alpha'\}$ with $\alpha' = \overline{\mathsf{req}}(\iota_b, n', u', p, ck', o')^{l'',\mu''}$ where
    - $\forall k \in [1 \ldots |\vec{v}|] : p(k) = v_k$
    - $ck' = get\_ck(M', u')$
    - $o' = (o = \mathrm{orig}(u))\ ?\ o : \bot$

    and hence have to show that $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha'$. We show that all the properties of Definition 9 are fulfilled.
    - Property 1 follows immediately from property 8 of Definition 9 for α
    - Property 2 follows immediately from property 8 of Definition 9 for α
    - Property 3 follows immediately from property 2 of Definition 11 for B
    - Property 4 is trivial since $\iota_u = \mathsf{usr}$
    - For property 5 we perform a case distinction:
      - If $u' \notin \mathcal{P}$, $I(\ell_a) \sqsubseteq_I o'$ or $\mu' = \mathsf{hon}$ then the claim is trivial.
      - If $u \in \mathcal{P}$, $I(\ell_a) \not\sqsubseteq_I o'$ and $\mu' = \mathsf{att}$ then we know that $I(\ell_a) \not\sqsubseteq_I \mathrm{orig}(u)$. We then know that the code at endpoint u can be typed with $b = \mathsf{hon}$ and we get by rule (T-REDIRECT) that $u \notin \mathcal{P}$. Since the redirect URL must appear as a constant in the code, we apply this result in any case and reach a contradiction.
  - Property 6 is trivial since high integrity pages only include high integrity pages.
  - For property 7 we distinguish two cases:
    - If $T = \{\}$ or $\mu' = \mathsf{att}$ the claim is trivial

∗ If $T = \{\}$ and $\mu' = \mathsf{hon}$, then we know that $\mu = \mathsf{hon}$ and $\mu'' = \mathsf{hon}$. The claim then follows using rule (T-REDIR)

□

We have now shown all lemmas for browser steps and move on to the server. First, we introduce typing for the server:

Definition 12 (Server Typing)**.** *Let $S = (D, \phi, t)$ be a server with $D = (D_@, D_\$)$. We write $\Gamma \vDash_{\ell_a,\mathsf{usr}} S$, if*

1. • *if $\iota_b = \mathsf{usr}$ then for all $i \in dom(D_@)$, for all $r \in dom(D_@(i))$ we have if $D_@(i)(r) = v^\tau$ then $\tau \sqsubseteq_{\ell_a} ref_\tau(\Gamma_{\mathcal{R}^@}(r))$*
   • *if $\iota_b \neq \mathsf{usr}$ then for all $i \in dom(D_@)$, for all $r \in dom(D_@(i))$ we have if $D_@(i)(r) = v^\tau$ then $\tau \sqsubseteq_{\ell_a} \ell_a$*
2. *for all $i \in dom(D_\$)$, for all $r \in dom(D_\$(j))$ we have if $D_\$(j)(r) = v^\tau$ then $\tau = ref_\tau(\Gamma_{\mathcal{R}^\$}(r))\overline{\sqcap} jlabel(j)$*
3. *for all $u \in urls(S)$, for all $j \in dom(\phi)$we have that $\rho(\phi(j), u) \sqsubseteq_{\ell_a} jlabel(j)$*
4. $\Gamma^0 \vDash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$
5. *For all $t \in threads(S)$ with $t = \lceil c \rfloor^{l,\mu}_{(n,u,\iota_b,o),(i,j)}$ we have if $\iota_b = \mathsf{usr}$, $u \in \mathcal{P}$ and $o \neq \bot$ and $I(\ell_a) \not\sqsubseteq_I o$ then $\mu = \mathsf{hon}$.*

Intuitively, according to Definition 12 a server is well-typed if

1. For the global memories we have that
   - for honest users, all values respect the typing environment
   - for the attacker, all values are of the attackers type ℓ_a
2. All values in session memories respect the typing environment (taking the label of the session identifier into account)
3. All sessions are protected by session identifiers whose security guarantees are stronger than the one of the passwords corresponding to the identity stored in the session.
4. All server threads are well-typed.
5. For all threads the integrity label is as least as low as the origin

We now show the same lemmas we showed for the browser on the server side , starting with the substitution of variables in server expressions.

Lemma 9 (Server Expression Substitution)**.** *Whenever we have* $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \tau$ *and we have a substitution* σ *with* $dom(\sigma) = dom(\Gamma_{\mathcal{X}})$ *and* $\forall x \in dom(\Gamma_{\mathcal{X}}).\, \sigma(x) = v_x^{\tau_x}$ *with* $\tau_x \sqsubseteq_{\ell_a} \Gamma_{\mathcal{X}}(x)$ *then for all* $\Gamma'_{\mathcal{X}}$, $(\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}), b \vDash^{\mathsf{se}}_{\ell_a} se\sigma : \tau$.

Proof. We perform an induction on the typing derivation of $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \tau$:

- (T-EVAR). Then $se = x$ and $se\sigma = v_x^{\tau_x}$ with $\tau_x \sqsubseteq_{\ell_a} \Gamma_{\mathcal{X}}(x)$. The claim follows directly from rule (T-EVAL) and (T-SUB).
- (T-ESESREF). Then $se = @r = se\sigma$ and the claim is trivial.
- (T-EGLOBREF). Then $se = \$r = se\sigma$ and the claim is trivial.
- (T-EVAL). Then $se = v^{\tau_v} = se\sigma$ and the claim is trivial.
- (T-EUNDEF). Then $se = \bot = se\sigma$ and the claim is trivial.
- (T-ENAME). Then $se = n^{\tau_n} = se\sigma$ and the claim is trivial.
- (T-EFRESH). Then $se = \mathit{fresh}()^{\tau} = se\sigma$ and the claim is trivial.
- (T-EBINOP) Then $se = se_1 \odot se_2$ with $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se_1 : \tau_1$ and $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se_2 : \tau_2$ and $\tau = \mathit{label}(\tau_1) \sqcup \mathit{label}(\tau_2)$. We also have $se\sigma = se_1\sigma \odot se_2\sigma$. By induction we know that $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se_1\sigma : \tau'_1$ and $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se_2\sigma : \tau'_2$ with $\tau'_1 \sqsubseteq_{\ell_a} \tau_1$ and $\tau'_2 \sqsubseteq_{\ell_a} \tau_2$. We then know that $\mathit{label}(\tau'_1) \sqcup \mathit{label}(\tau'_2) \sqsubseteq_{\ell_a} \mathit{label}(\tau_1) \sqcup \mathit{label}(\tau_2) = \tau$, and the claim follows by (T-BINOP) and (T-BESUB).
- (T-ESUB) follows by induction and by the transitivity of $\sqsubseteq_{\ell_a}$.

□

To show the substitution lemma for server commands we first need to show auxiliary lemmas that deal with the program counter.

First, we show that whenever server code can be typed with a $\mathtt{pc}$, it can also be typed with any $\mathtt{pc}$ of higher integrity.

Lemma 10 (Server Program Counter Substitution)**.** *Whenever we have* $\Gamma, \ell_s, pc \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell_s', pc'$ *and* $pc^* \sqsubseteq_I pc$ *then* $\Gamma, \ell_s, pc^* \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c\sigma : \ell_s, pc^{**}$ *with* $pc^{**} \sqsubseteq_I pc'$.

Proof. We perform the proof by induction on the typing derivation

- (T-SKIP) The claim is trivial
- (T-LOGIN) The claim follows from the transitivity of $\sqsubseteq_I$
- (T-START) The claim is trivial

- (T-SETGLOBAL) The claim follows from the transitivity of $\sqsubseteq_I$
- (T-SETSESSION) The claim follows from the transitivity of $\sqsubseteq_I$
- (T-SEQ) The claim follows by induction on the two subcommands.
- (T-IF): Then $c = \textbf{if } se \textbf{ then } c_1 \textbf{ else } c_2$ with
  - $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \tau$.
  - $\mathtt{pc}' = \mathtt{pc} \sqcup_I I(\tau)$
  - $\Gamma, \ell_s, \mathtt{pc}'' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_1 : {\ell_s}'', \mathtt{pc}_1$
  - $\Gamma, \ell_s, \mathtt{pc}'' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_2 : {\ell_s}''', \mathtt{pc}_2$
  - $\mathtt{pc}'' =$ $((c \text{ and } c' \text{ do not contain } \textbf{reply}, \textbf{redir}, \textbf{tokencheck} \text{ or } \textbf{origincheck})\ ?\ \mathtt{pc} : \mathtt{pc}')$ $\sqcup_I \mathtt{pc}_1 \sqcup_I \mathtt{pc}_2$

  Let $\mathtt{pc}''' = \mathtt{pc}^* \sqcup_I I(\tau)$, then $\mathtt{pc}''' \sqsubseteq_I \mathtt{pc}'$ and we can apply the induction hypothesis for c_1 and c_2 and get
  - $\Gamma, \ell_s, \mathtt{pc}''' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_1 : {\ell_s}'', \mathtt{pc}_1^*$
  - $\Gamma, \ell_s, \mathtt{pc}''' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_2 : {\ell_s}''', \mathtt{pc}_2^*$

  with $\mathtt{pc}_1^* \sqsubseteq_I \mathtt{pc}_1$ and $\mathtt{pc}_2^* \sqsubseteq_I \mathtt{pc}_2$ and the claim follows by applying (T-IF).
- (T-TCHECK) The claim is trivial
- (T-PRUNETCHECK) The claim is trivial
- (T-OCHCK) The claim is trivial
- (T-PRUNEOCHCK) The claim is trivial
- (T-REPLY) With $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{t}, l_r$, we let $\mathtt{pc}' = \mathtt{pc} \sqcup_I l_r$ and $\mathtt{pc}'' = \mathtt{pc} \sqcup_I l_r$. We do a case distinction on b:
  - If $b =$ hon we get $\mathtt{pc} \sqsubseteq_I l_r$, hence $\mathtt{pc}' = l_r$ and because of of $\mathtt{pc}' \sqsubseteq_I \mathtt{pc}$ we also get $\mathtt{pc}'' = l_r$ and the claim follows.
  - If $b \neq$ hon we have $\mathtt{pc} = \top_I$ We hence also have $\mathtt{pc}^* = \top_I$ and the claim follows.
- (T-REDIR) The claim follows from the transitivity of $\sqsubseteq_I$
- (T-RESET) The claim is trivial.

□

We now show that if server code containing variables is well-typed in a typing environment typing these variables, then the code is also well-typed after instantiating these variables with concrete values of the same type.

Lemma 11 (Server Substitution)**.** *Whenever we have* $\Gamma, \ell_s, pc \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell_s', pc^*$ *and we have a substitution* σ *with* $dom(\sigma) = dom(\Gamma_{\mathcal{X}})$ *and* $\forall x \in dom(\Gamma_{\mathcal{X}}).\, \sigma(x) = v_x^{\tau_x}$ *with* $\tau_x \sqsubseteq_{\ell_a} \Gamma_{\mathcal{X}}(x)$ *then for all* $\Gamma'_{\mathcal{X}}$*, we have* $\Gamma', \ell_s, pc \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c\sigma : \ell_s', pc^{**}$ *with* $\Gamma' = (\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}})$.

Proof. We do the proof by induction on the typing derivation.

- (T-BSEQ): Then $c = c_1, c_2$. The claim follows by applying the induction hypothesis to s_1 and s_2 using Lemma 10 and applying rule (T-SEQ)
- (T-SKIP): The claim follows trivially.
- (T-SETSESSION): The claim follows from Lemma 9 and the transitivity of $\sqsubseteq_{\ell_a}$.
- (T-SETGLOBAL): The claim follows from Lemma 9 and the transitivity of $\sqsubseteq_{\ell_a}$.
- (T-LOGIN): The claim follows from Lemma 9 and the transitivity of $\sqsubseteq_{\ell_a}$.
- (T-START): The claim follows from Lemma 9 and the transitivity of $\sqsubseteq_{\ell_a}$.
- (T-IF): Then $c = \mathbf{if}\ se\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$ with
  - $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \tau$.
  - $\mathtt{pc}' = \mathtt{pc} \sqcup_I I(\tau)$
  - $\Gamma, \ell_s, \mathtt{pc}' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_1 : \ell_s'', \mathtt{pc}_1$
  - $\Gamma, \ell_s, \mathtt{pc}' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_2 : \ell_s''', \mathtt{pc}_2$

  By induction we know
  - $\Gamma', \ell_s, \mathtt{pc}' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_1\sigma : \ell_s'', \mathtt{pc}_1^*$
  - $\Gamma', \ell_s, \mathtt{pc}' \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_2\sigma : \ell_s''', \mathtt{pc}_2^*$

  By Lemma 9 we know that $\Gamma', \ell_s \vDash^{\mathsf{se}}_{\ell_a} se\sigma : \tau$ The claim then follows by applying rule (T-IF).
- (T-AUTH) The claim follows from Lemma 9 and the transitivity of $\sqsubseteq_{\ell_a}$.
- (T-PRUNETCHECK) The claim follows from Lemma 9 and the fact that there is no subtyping on credentials of high confidentiality.
- (T-OCHCKSUCC) The claim follows trivially.
- (T-OCHCKFAIL) The claim follows trivially.

- (T-TCHECK) The claim follows from Lemma 9.
- (T-REPLY) Let variables be assigned as in the rule. The claim then follows by applying Lemma 9 for all se_k. The claim then follows immediately.
- (T-REDIR) Let variables be assigned as in the rule. The claim then follows by applying Lemma 9 for all se_k.

□

We now show that typing of server expressions is preserved under evaluation.

Lemma 12 (Server Expression Typing)**.** *Let $S = (D, \phi, t)$ be a server with $\Gamma \vDash_{\ell_a,\mathsf{usr}} S$ and let $\lceil c \rfloor^{l,\mu}_{R,i,j} \in running(S)$. Then for any server expression se, if $\Gamma, jlabel(j) \vDash^{\mathsf{se}}_{\ell_a} se : \tau$ then $\Gamma, jlabel(j) \vDash^{\mathsf{se}}_{\ell_a} eval_{i,j}(se, D) : \tau$.*

Proof. Proof by induction over the expression se.

- $se = v^{\tau_v}$: Then $eval_{i,j}(se, D) = se$ and the claim is trivial.
- $se = se_1 \odot se_2$: By induction analog to case in in Lemma 5.
- $se = @r$: straightforward from property 1 of Definition 12 using (T-EGLOBREF) and (T-ESUB)
- $se = \$r$: straightforward from property 2 of Definition 12 using (T-ESESREF) and (T-ESUB)
- $se = fresh()^{\tau_f}$: straightforward from (SE-FRESH), (T-FRESH) and (T-ENAME)

□

Next, we show that whenever a server thread is typable with the session label $\times$, then it is also typable with any other session label.

Lemma 13 (Server Typing with $\ell_s = \times$)**.** *Whenever we have $\Gamma, \times, pc \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell_s{}', pc$ then we also have $\Gamma, \ell_s, pc \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell_s{}'', pc$ for all ℓ_s, where $\ell_s{}' = \times$ or $\ell_s{}' = \ell_s{}''$.*

Proof. This is simple by inspecting the typing rules and the observation that $\ell_s = \times$ implies that the session memory cannot be used. Hence the code that is typed with $\ell_s = \times$ can be typed with any session label. Furthermore, if the session label is set to a different label during typing, this is unaffected by the old session label. □

We are now ready to show that whenever a well-typed server takes an internal step, it results in another well-typed server.

Lemma 14 (Server Subject Reduction)**.** *Let S be a server with $\Gamma^0 \vDash_{\ell_a,\mathsf{usr}} S$ and $S \xrightarrow{\alpha} S'$, where $\alpha \in \{\bullet, \sharp[\vec{v}]_\ell^{\iota_b,\iota_u}\}$ Then we have $\Gamma^0 \vDash_{\ell_a,\mathsf{usr}} S'$.*

Proof. Let $S = (D, \phi, t)$ and let $S' = (D', \phi', t')$ Then there exists $\lceil c \rfloor_{R,i,j}^{l,\mu} \in \text{running}(S)$ with $(D, \phi, \lceil c \rfloor_{R,i,j}^{l,\mu}) \xrightarrow{\alpha} (D', \phi', \lceil c' \rfloor_{R,i,j'}^{l',\mu})$.

Because of rules (S-LPARALLEL), (S-RPARALLEL) and (T-PARALLEL) it is sufficient to show $\Gamma^0 \vDash_{\ell_a,\mathsf{usr}} (D, \phi, \lceil c' \rfloor_{R,i,j'}^{l',\mu})$, assuming $\Gamma^0 \vDash_{\ell_a,\mathsf{usr}} (D, \phi, \lceil c \rfloor_{R,i,j}^{l,\mu})$.

We chose b, ℓ_{s1}, $\mathtt{pc}_1$ and Γ as in rule (T-RUNNING):

$$\text{Let } b = \begin{cases} \mathsf{hon} & \text{if } \mu = \mathsf{hon} \wedge \iota_b = \mathsf{usr} \\ \mathsf{csrf} & \text{if } \mu = \mathsf{att} \wedge \iota_b = \mathsf{usr} \\ \mathsf{att} & \text{if } \iota_b \neq \mathsf{usr} \end{cases} \text{ and let } \ell_{s1} = jlabel(j) \text{ and let } \mathtt{pc}_1 = l.$$

With $\Gamma^0 = (\Gamma_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}})$ and $\Gamma'_{\mathcal{R}^@} = (\iota_b = \mathsf{usr})\ ?\ \Gamma_{\mathcal{R}^@}\ :\ \{\_ \mapsto \ell_a\}$ we let $\Gamma = (\Gamma_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma'_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}})$.

We furthermore let $\ell_{s2} = jlabel(j')$ and $\mathtt{pc}_2 = l'$.

We now show that S' fulfills all properties of Definition 12.

However, for property 4 of Definition 12 we will show the following stronger claim:

Whenever $\Gamma, \ell_{s1}, \mathtt{pc}_1 \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell'_{s1}, \mathtt{pc}'_1$ we have $\Gamma, \ell_{s2}, \mathtt{pc}_2 \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c' : \ell'_{s2}, \mathtt{pc}'_2$ where $\mathtt{pc}'_2 \sqsubseteq_I \mathtt{pc}'_1$ and $\ell'_{s1} = \times$ or $\ell'_{s1} = \ell'_{s2}$

For all cases property 5 is trivial.

We perform the proof by induction the step taken.

- (S-SKIP). This case is trivial.
- (S-SEQ) Then we know
  - $c = c_1; c_2$
  - $(D, \phi, \lceil c_1 \rfloor_{R,i,j}^{l,\mu}) \xrightarrow{\alpha} (D', \phi', \lceil c'_1 \rfloor_{R,i,j'}^{l',\mu})$
  - $c' = c'_1; c_2$.

  All properties of Definition 12 except for property 4 follow immediately by the induction hypothesis applied to c_1.

  By rule (T-SEQ) we know that for some $\mathtt{pc}''_1$ and ℓ''_{s1}
  - $\Gamma, \ell_{s1}, \mathtt{pc}_1 \vDash^{\mathsf{c}}_{\ell_a,C} c_1 : \ell''_{s1}, \mathtt{pc}''_1$
  - $\Gamma, \ell''_{s1}, \mathtt{pc}''_1 \vDash^{\mathsf{c}}_{\ell_a,C} c_2 : \ell'_{s1}, \mathtt{pc}'_1$

  By induction we know that for some $\mathtt{pc}_2$,$\mathtt{pc}'_2$,ℓ_{s2},ℓ'_{s2}

- $\Gamma, \ell_{s2}, \mathtt{pc}_2 \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c_1 : \ell_{s2}', \mathtt{pc}_2'$
- $\mathtt{pc}_2' \sqsubseteq_I \mathtt{pc}_1''$
- $\ell_{s1}'' = \times$ or $\ell_{s1}'' = \ell_{s2}'$

Using Lemma 10 we get $\Gamma, \ell_{s1}'', \mathtt{pc}_2' \vDash^{\mathsf{c}}_{\ell_a,C} c_2 : \ell_{s1}', \mathtt{pc}_2''$ with $\mathtt{pc}_2'' \sqsubseteq_I \mathtt{pc}_1'$

Using Lemma 13 we furthermore get $\Gamma, \ell_{s2}', \mathtt{pc}_2' \vDash^{\mathsf{c}}_{\ell_a,C} c_2 : \ell_{s2}'', \mathtt{pc}_2''$ with $\ell_{s1}' = \times$ or $\ell_{s1}' = \ell_{s2}''$

Using (T-SEQ) we can then conclude $\Gamma, \ell_{s2}, \mathtt{pc}_2 \vDash^{\mathsf{c}}_{\ell_a,C} c_1'; c_2 : \ell_{s2}'', \mathtt{pc}_2''$ and the claim follows.

- (S-IFTRUE) then
  - $c = \textbf{if } se \textbf{ then } c_1 \textbf{ else } c_2$
  - $eval_{i,j}(se, D) = true^{\tau}$
  - $j' = j$
  - $l' = l \sqcup_I I(\tau)$
  - $c' = (\textbf{reply}, \textbf{redir}, \textbf{tokencheck}, \textbf{origincheck} \in coms(c)) \; ? \; c_1 \; : \; c; \mathsf{reset}\ l$

  All properties of Definition 12 except for property 4 follow immediately by the induction hypothesis applied to c_1 using rules (S-SEQ) and (S-RESET).

  By (T-IF) we know
  - $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \tau'$
  - $\mathtt{pc}' = \mathtt{pc}_1 \sqcup_I I(\tau)$
  - $\Gamma, \ell_{s1}, \mathtt{pc}' \vDash^{\mathsf{c}}_{\ell_a,C} c_1 : \ell_{s2}', \mathtt{pc}_2'$ for some $\ell_{s2}', \mathtt{pc}_2'$
  - $\Gamma, \ell_{s1}, \mathtt{pc}' \vDash^{\mathsf{c}}_{\ell_a,C} c_2 : \ell_{s2}'', \mathtt{pc}_2''$ for some $\ell_{s2}'', \mathtt{pc}_2''$
  - $\ell_{s1}' = \ell_{s2}'$ or $\ell_{s1}' = \times$
  - $\mathtt{pc}_1' = \textbf{reply}, \textbf{redir}, \textbf{tokencheck}, \textbf{origincheck} \in coms(c) \; ? \; \mathtt{pc}_2' \sqcup_I \mathtt{pc}_2'' \; : \; \mathtt{pc}_1$

  By Lemma 12 we know that $\tau \sqsubseteq_{\ell_a} \tau'$. Hence $\mathtt{pc}_2 = l' \sqsubseteq_I \mathtt{pc}'$. We thus have by Lemma 10 that $\Gamma, \ell_{s2}, \mathtt{pc}_2 \vDash^{\mathsf{c}}_{\ell_a,C} c_1 : \ell_{s2}', \mathtt{pc}_2^*$ with $\mathtt{pc}_2 * \sqsubseteq_I \mathtt{pc}_2'$.

  If $\textbf{reply}, \textbf{redir}, \textbf{tokencheck}, \textbf{origincheck} \in coms(c)$, the claim follows immediately.

  Otherwise, we using (T-SEQ) and (T-RESET) we observe, that $\Gamma, \ell_{s2}, \mathtt{pc}_2 \vDash^{\mathsf{c}}_{\ell_a,C} c_1; \mathsf{reset}\ l : \ell_{s2}', l$. With $\mathtt{pc}_1' = \mathtt{pc}_1 = l = \mathtt{pc}_2'$ the claim follows immediately.
- (S-IFFALSE) then the claim follows analog to the previous one.
- (S-TCTRUE) Then
  - $c = \textbf{if tokenchk}(se, se') \textbf{ then } c'$
  - $eval_{i,j}(se, D) = v_1^{\tau_1}$

- $eval_{i,j}(se', D) = v_1^{\tau_2}$
- $v_1 = v_2$
- $l' = l$
- $j' = j$

All properties of Definition 12 except for property 4 are trivial.

We know that typing was done using rule (T-TCHK) or (T-PRUNETCHK).

We want to show that (T-TCHK) was used. To this end, we assume that (T-PRUNECHK) was used and show a contradiction.

By rule (T-TCHKPRUNE) we know

- $se = x$ for some x and $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} x : \tau_1'$
- $se' = r$ for some r and $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} r : \tau_2'$
- $\tau_2 = \mathtt{cred}(\ell)$
- $\tau_1 \neq \mathtt{cred}(\ell)$
- $\mathtt{cred}(\ell) \not\sqsubseteq_C C(\ell_a)$

By Lemma 12 we know that $\tau_1 \sqsubseteq_{\ell_a} \tau_1'$ and $\tau_2 \sqsubseteq_{\ell_a} \tau_2'$. By the definition of $\sqsubseteq_{\ell_a}$, we know that $\tau_1 = \tau_1'$. Since the set of credentials at label $label(\tau_1)$ is disjoint from the set of the set of any other values, and since $v_1 = v_2$, we know that also $\tau_1 = \tau_2$. Using the definition of $\sqsubseteq_{\ell_a}$ we get $\tau_2 = \tau_2'$. We hence have $\tau_1' = \tau_2'$ which contradicts the assumption.

We thus know that (T-TCHK) and we get $\Gamma, \ell_{s1}, \mathtt{pc}_1 \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c' : \ell_{s1}', \mathtt{pc}_1'$ and the claim follows.

- (S-TCFALSE) Then
  - $c = \textbf{if tokenchk}(se, se')\ \textbf{then}\ c''$
  - $c' = \textbf{reply}\ (\mathsf{error}, \textbf{skip}, \{\})$
  - $eval_{i,j}(se, D) = v_1^{\tau_1}$
  - $eval_{i,j}(se', D) = v_1^{\tau_2}$
  - $v_1 \neq v_2$
  - $l' = l$
  - $j' = j$

  All properties of Definition 12 except for property 4 are trivial.

  By (T-REPLY) we immediately get $\Gamma, \ell_{s1}, \mathtt{pc}_1 \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \textbf{reply}\ (\mathsf{error}, \textbf{skip}, \{\}) : \ell_{s1}, \mathtt{pc}_1$.
- (S-RESET) All properties of Definition 12 are trivial, where property 4 follows immediately from (T-RESET).
- (S-RESTORESESSION) We have

- $c = \textbf{start}\ se$
- $c' = \textbf{skip}$
- $eval_{i,j}(se, D) = v^\tau$
- $v \in dom(D_\$)$
- $l' = C(\tau) \sqsubseteq_C C(\ell_a)\ ?\ \bot\ :\ label(\tau)$
- $j' = v$

All properties of Definition 12 except for property 4 are trivial.

By (T-START) we get

- $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \texttt{cred}(\ell)$
- $\ell_{s1}' = (C(\texttt{cred}(\ell)) \sqsubseteq_C C(\ell_a))\ ?\ (\bot_C, \top_I) : \ell$

By Lemma 12 we know that $\tau \sqsubseteq_{\ell_a} \texttt{cred}(\ell)'$ We distinguish two cases:

- If $C(\texttt{cred}(\ell)) \sqsubseteq_C C(\ell_a)$ then also $C(\tau) \sqsubseteq_C C(\ell_a)$ and we have $\ell_{s1}' = (\bot_C, \top_I) = jlabel(j') = \ell_{s2}'$.
- If $C(\texttt{cred}(\ell)) \not\sqsubseteq_C C(\ell_a)$ we know $\tau = \texttt{cred}(\ell)$ and we have $\ell_{s1}' = \ell = jlabel(j') = \ell_{s2}'$.

- (S-NEWSESSION) We immediately get property 1, 3 of Definition 12. Property 2 follows immediately using rule (T-EUNDEV) since the freshly created memory is empty. Property 4 follows analog to the previous case.

- (S-SETGLOBAL) We immediately get property 2, 3, 4 of Definition 12, using (T-SETGLOBAL).

  We have $c = @r := se$, $eval_{i,j}(se, D) = v^\tau$ and $\tau' = \tau \sqcup \text{ref}_\tau(\Gamma'_{\mathcal{R}^@}(r)) \tilde{\sqcup}_I l$ with $\Gamma'_{\mathcal{R}^@} = (\iota_b = \mathsf{usr})\ ?\ \Gamma_{\mathcal{R}^@}\ :\ \{\_ \mapsto \ell_a\}$.

  We know using rule (T-SETGLOBAL) that

  - $\Gamma', @r \vDash^{\mathsf{sr}}_{\ell_a} \texttt{ref}(\tau') :$
  - $\Gamma', \ell_{s1} \vDash^{\mathsf{se}}_{\ell_a} se : \tau'$
  - $\texttt{pc}_1 \sqsubseteq_I I(\tau')$

  Using (T-GLOBREF) and (T-REFSUB) we know that $\tau' \sqsubseteq_{\ell_a} \text{ref}_\tau(\Gamma'_{\mathcal{R}^@}(r))$. Using Lemma 12 we know $\tau \sqsubseteq_{\ell_a} \tau'$.

  We hence know that $\tau' = \text{ref}_\tau(\Gamma'_{\mathcal{R}^@}(r))$ and property 1 follows.

- (S-SETSESSION) We immediately get property 1, 3, 4 of Definition 12, using rule (T-SETSESSION).

  We have $c = \$r := se$, $eval_{i,j}(se, D) = v^\tau$ and $\tau' = \tau \sqcup (\text{ref}_\tau(\Gamma_{\mathcal{R}^\$}(r)) \tilde{\sqcup} jlabel(j)) \tilde{\sqcup}_I l$.

  We know using rule (T-SETSESSION) that

- $\Gamma, \$r \vDash^{\mathsf{sr}}_{\ell_a} \mathtt{ref}(\tau') :$
- $\Gamma, \ell_{s1} \vDash^{\mathsf{se}}_{\ell_a} se : \tau'$
- $\mathtt{pc}_1 \sqsubseteq_I I(\tau')$

Using (T-SESREF) and (T-REFSUB) we know that $\tau' \sqsubseteq_{\ell_a} (\mathrm{ref}_\tau(\Gamma_{\mathcal{R}^\$}(r))\tilde{\sqcup}\ell_{s1})$. Using Lemma 12 we know $\tau \sqsubseteq_{\ell_a} \tau'$.

We hence know that $\tau' = \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^\$}(r))\tilde{\sqcup} jlabel(j)$ and the property 2 follows.

- (S-LOGIN) We immediately get property 1, 2, 4 of Definition 12. Property 3 follows from rule (T-LOGIN).
- (S-AUTH) All properties are trivial
- (S-OCHCKSUCC) Then
  - $c = \mathbf{if\ originchk}(L)\ \mathbf{then}\ c'$
  - $R = n, u, \iota_b, o$
  - $o \in L$

  All properties of Definition 12 except for property 4 are trivial.

  We know that typing was done using rule (T-OCHK) or (T-PRUNEOCHK).

  We want to show that (T-OCHK) was used. To this end, we assume that (T-PRUNECHK) was used and show a contradiction

  By rule (T-OCHKPRUNE) we know
  - $\forall l \in L. I(\ell_a) \not\sqsubseteq_I l$
  - $u \in \mathcal{P}$
  - $b = \mathsf{csrf}$

  We hence have $I(\ell_a) \not\sqsubseteq_I o$.

  Then by 5 of Definition 12, we know that $\mu = \mathsf{hon}$, which is an immediate contradiction.
- (T-OCHCKFAIL) This case is analog to the case of rule (T-TCHKCFAIL)

□

We now show that any expression that is well-typed in an honest typing branch is also well-typed when typing in the attacker's setting and that all expressions have type ℓ_a in the attacked setting.

Lemma 15 (Attacker Server Expression Typability)**.** *For all server expressions se we have if*

- $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \tau$

- $\forall x \in \vec{x}.\, \Gamma'_{\mathcal{X}}(x) = \ell_a$
- $\forall r \in \mathcal{R}.\, \Gamma'_{\mathcal{R}^{@}}(r) = \texttt{ref}(\ell_a)$
- $\ell_s \neq \times \Rightarrow {\ell_s}' = \ell_a$
- $se = \overline{se'}$ *for some* se'

then we have $(\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma'_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}), {\ell_s}' \vDash^{\mathsf{se}}_{\ell_a} se : \ell_a$

Proof. We prove the claim by induction over the typing derivation for $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \tau$

- (T-EVAL) Since $se = \overline{se'}$ from some se' , we have $se = v^{\perp}$. The claim then follows since $\perp \sqsubseteq_{\ell_a} \ell_a$.
- (T-EFRESH) Then we have $se = \mathit{fresh}()^{\tau}$. The claim is trivial because of $b = \mathsf{att}$.
- (T-VUNDEF) Trivial.
- (T-EVAR) Follows immediately from the definition of $\Gamma'_{\mathcal{X}}$.
- (T-EGLOBREF) Follows immediately from the definition of $\Gamma'_{\mathcal{R}^{@}}$.
- (T-ESESREF) Then we know that $\ell_s \neq \times$ and hence ${\ell_s}' = \ell_a$. The claim then follows immediately from (T-ESESREF) and (T-ESUB)
- (T-EBINOP) Then the claim follows immediately by induction.
- (T-ESUB) The claim follows immediately by induction.

□

Next we show, that any server thread that is well-typed in the honest setting is also well-typed when typing in the attacker's setting.

Lemma 16 (Attacker Server Typability)**.** *Let* t *be a thread with*

- $t = u[\vec{r}](\vec{x}) \hookrightarrow c$ *with* $\Gamma^0 \vDash^{\mathsf{t}}_{\ell_a, \mathcal{P}} t$
- $\forall x \in \vec{x}.\Gamma_{\mathcal{X}}(x) = \ell_a$
- $\forall r \in \mathcal{R}.\Gamma_{\mathcal{R}^{@}}(r) = \ell_a$
- $t = \overline{t'}$ *for some* t'

we have $(\Gamma^0_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma_{\mathcal{R}^{@}}, \Gamma^0_{\mathcal{R}^{\$}}, \Gamma^0_{\mathcal{V}}), \times, \top_I \vDash^{\mathsf{c}}_{\ell_a, (u, \mathsf{att}, \mathcal{P})} c : \_, \_$

Proof. By $\Gamma^0 \vDash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$ we know by (T-RECV) that with $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{\tau}, l_r$ and $m = |\vec{x}|$ and $\Gamma^h_{\mathcal{X}} = x_1 \colon \tau_1, \ldots, x_m \colon \tau_m$ we have

$$(\Gamma^0_{\mathcal{U}}, \Gamma^h_{\mathcal{X}}, \Gamma^0_{\mathcal{R}^@}, \Gamma^0_{\mathcal{R}^\$}, \Gamma^0_{\mathcal{V}}), \times, I(\ell_u) \vDash^{\mathsf{c}}_{\ell_a,(u,\mathsf{hon},\mathcal{P})} c : \_, \_$$

We let $\Gamma = (\Gamma^0_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma_{\mathcal{R}^@}, \Gamma^0_{\mathcal{R}^\$}, \Gamma^0_{\mathcal{V}})$ and now show the following stronger claim: Whenever

$$(\Gamma^0_{\mathcal{U}}, \Gamma^h_{\mathcal{X}}, \Gamma^0_{\mathcal{R}^@}, \Gamma^0_{\mathcal{R}^\$}, \Gamma^0_{\mathcal{V}}), \ell_{sh}, \mathtt{pc}_h \vDash^{\mathsf{c}}_{\ell_a,(u,\mathsf{hon},\mathcal{P})} c : \ell'_{sh}, \mathtt{pc}'_h$$

then

$$\Gamma, \ell_s, \top_I \vDash^{\mathsf{c}}_{\ell_a,(u,\mathsf{att},\mathcal{P})} c : \ell_s{}', \top_I$$

where

- $\ell_{sh} = \times \Rightarrow \ell_s = \times \wedge \ell_{sh} \neq \times \Rightarrow \ell_s = \ell_a$ and
- $\ell'_{sh} = \times \Rightarrow \ell_s{}' = \times \wedge \ell'_{sh} \neq \times \Rightarrow \ell_s{}' = \ell_a$ and

The proof is by induction on the honest typing derivation for c

- (T-SKIP) The claim is trivial.
- (T-SEQ) The claim follows directly from the induction hypothesis on the two subcommands.
- (T-IF) We have $c = \textbf{if } se \textbf{ then } c_t \textbf{ else } c_f$. With $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : \ell_a$ by Lemma 15. We have $\mathtt{pc}' = \top_I \sqcup_I I(\ell_a) = \top_I = \mathtt{pc}$ (in rule (T-IF)) and the claim follows from the induction hypothesis for c_t and c_f.
- (T-LOGIN) By Lemma 16, we get that all expressions are of type ℓ_a. Using rule (T-ESUB) we can also treat them as expressions of type $\mathtt{cred}((\bot_C, \top_I))$. The claim then follows immediately using (T-LOGIN).
- (T-START) By Lemma 16 we get that all expressions are of type ℓ_a. Using rule (T-ESUB) we can also treat them as expressions of type $\mathtt{cred}((\bot_C, \top_I))$. The claim then follows immediately using (T-START)
- (T-SETGLOBAL) We have $c = @r := se$ with $\Gamma, \ell_s{}' \vDash^{\mathsf{se}}_{\ell_a} se : \ell_a$ by Lemma 15 and $\Gamma_{\mathcal{R}^@}(r) = \mathtt{ref}(\ell_a)$. Using subtyping we can show $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se : (\bot_C, \top_I)$ and $\Gamma, \ell_s \vDash^{\mathsf{sr}}_{\ell_a} @r : \mathtt{ref}((\bot_C, \top_I))$ and the claim follows using rule (T-SETGLOBAL).
- (T-SETSESSION) We have $c = \$r := se$. The claim follows analogous to the previous one, using that $\Gamma, \ell_s \vDash^{\mathsf{sr}}_{\ell_a} \$r : \mathtt{ref}(\ell_a)$ because of $\ell_s = \ell_a$.
- (T-PRUNETCHECK): Impossible since this rule cannot be applied for $b = \mathsf{hon}$

- (T-TOKENCHECK): By Lemma 16 we get that all expressions are of type ℓ_a. Using rule (T-ESUB) we can also treat them as expressions of type $\texttt{cred}((\bot_C, \top_I))$. The claim then follows by induction and using (T-TOKENCHECK)
- (T-PRUNEOCHK) Impossible since this rule cannot be applied for $b = \mathsf{hon}$
- (T-OCHK) The claim follows immediately by induction.
- (T-AUTH): Then the claim follows immediately using rule (T-AUTHATT).
- (T-REPLY) From Lemma 16 and rule (T-ESUB) we know that for all variables x in the freshly generated environment $\Gamma'_{\mathcal{X}}$ we have $\Gamma'_{\mathcal{X}}(x) = (\bot_C, \top_I)$. Furthermore, with subtyping we can show $\Gamma, \ell_s \vDash^{\mathsf{sr}}_{\ell_a} r : \texttt{ref}((\bot_C, \top_I))$ for all $r \in \mathit{dom}(ck)$. The claim then follows immediately.
- (T-REDIR) This case follows analog to the previous case.

□

Next we show that whenever a server receives a well-formed request, the resulting running thread is also well-typed.

Lemma 17 (Server Request)**.** *Whenever a server* $S \xrightarrow{\alpha} S'$ *with* $\Gamma \vDash_{\ell_a,\mathsf{usr}} S$, $\alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{l,\mu}$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *then* $\Gamma \vDash_{\ell_a,\mathsf{usr}} S'$

Proof. Let $S = (D, \phi, t)$ and $S' = (D', \phi', t')$. We show that S' fulfills the properties of Definition 12. Property 2 and 3 follow immediately from rule (S-RECV) since the session memory and the trust mapping do not change.

- For property 1 we perform a case distinction
  - if $\iota_b \neq \mathsf{usr}$ then property 1 follows from property 4 of Definition 9.
  - if $\iota_b = \mathsf{usr}$ then property 1 follows from property 3 of Definition 9.
- For property 4 ,because of $S \xrightarrow{\alpha} S'$ we know $(D, \phi, u[\vec{r}](\vec{x}) \hookrightarrow c) \xrightarrow{\alpha} (D', \phi', \lceil c\sigma \rfloor^{l,\mu}_{n,u,\iota_b,i,\bot} \parallel u[\vec{r}](\vec{x}) \hookrightarrow c)$. It is hence sufficient, because of rule (T-PARALLEL), to show $\Gamma^0 \vDash^{\mathsf{t}}_{\ell_a,\mathcal{P}} \lceil c\sigma \rfloor^{l,\mu}_{(n,u,\iota_b),(i,\bot)}$

  We perform a case distinction:
  - if $\iota_b \neq \mathsf{usr}$ then by rule (T-RUNNING) with $b = \mathsf{att}$ and $\ell_s = \mathit{jlabel}(\bot) = \times$, $\Gamma'_{\mathcal{R}^@} = \{\_ \mapsto \ell_a\}$ we have to show

    $$(\Gamma_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma'_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}}), \ell_s, l \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c\sigma : \ell_s', l$$

    Because of Lemma 16 we get with $\Gamma'_{\mathcal{X}} = x_1 : \ell_a \cdots x_m : \ell_a$

$$(\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma'_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}), \ell_s, \top_I \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c : \ell_s{}', \top_I$$

With property 2 of Definition 9 we can use Lemma 11 for the substitution σ and the claim follows using Lemma 10.

– if $\iota_b = \mathsf{usr} \wedge \mu = \mathsf{att}$ then by rule (T-RUNNING) with $b = \mathsf{csrf}$ and $\ell_s = jlabel(\bot) = \times$, we have to show $\Gamma, \ell_s, l \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c\sigma : \ell_s{}', l$.
Since $l \sqsubseteq_I \top_I$ using Lemma 10 it is sufficient to show

$$\Gamma, \ell_s, \top_I \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c\sigma : \ell_s{}', \top_I$$

From rule (T-RECV) we get with $\Gamma'_{\mathcal{X}} = x_1 \colon (\bot_C, \top_I), \ldots, x_m \colon (\bot_C, \top_I)$ that

$$(\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}), \times, I(\ell_a) \vDash^{\mathsf{c}}_{\ell_a,(u,\mathsf{csrf},\mathcal{P})} c : \_, I(\ell_a)$$

With property 2 of Definition 9 we can use Lemma 11 for the substitution σ and the claim follows.

– if $\iota_b = \mathsf{usr} \wedge \mu = \mathsf{hon}$ then by rule (T-RUNNING) with $b = \mathsf{hon}$ and$\ell_s = jlabel(\bot) = \times$, we have to show $\Gamma, \ell_s, l \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c\sigma : \ell_s{}', l$.
Since we know $l \sqsubseteq_I I(\ell_u)$ by property 1 of Definition 9, using Lemma 10 it is sufficient to show

$$\Gamma, \ell_s, I(\ell_u) \vDash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} c\sigma : \ell_s{}', I(\ell_u)$$

From rule (T-RECV) we get with $\Gamma_{\mathcal{U}}(u) = \ell_u, \vec{t}, lr$ and $\Gamma'_{\mathcal{X}} = \Gamma^0_{\mathcal{X}}, x_1 \colon t_1, \ldots, x_m \colon t_m$

$$(\Gamma_{\mathcal{U}}, \Gamma'_{\mathcal{X}}, \Gamma_{\mathcal{R}^{@}}, \Gamma_{\mathcal{R}^{\$}}, \Gamma_{\mathcal{V}}), \times, I(\ell_a) \vDash^{\mathsf{c}}_{\ell_a,(u,\mathsf{hon},\mathcal{P})} c : \_, I(\ell_a)$$

With property 1 of Definition 9 we can use Lemma 11 for the substitution σ and the claim follows.

- Property 5 follows immediately from property 5 of Definition 9.

□

We now show that all responses by the server fulfill these conditions.

Lemma 18 (Server Response)**.** *Whenever a server* $S \xrightarrow{\alpha} S'$ *with* $\Gamma \vDash_{\ell_a,\mathsf{usr}} S$, $\alpha = \overline{\mathsf{res}}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l,\mu}$ *then* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} S'$

Proof. $\Gamma \vDash_{\ell_a,\mathsf{usr}} S'$ is trivial in all cases. We show that α fulfills all properties of Definition 10. We perform a case distinction on the rule used to type the reply.

- (T-REDIR): Property 1 follows directly from (T-REDIR) We perform a case distinction:

  - If $\iota_b \neq \mathsf{usr}$ Then we need to show property 2 which follows immediately from the typing rule, using Lemma 15
  - If $\iota_b = \mathsf{usr}$ and $\mu = \mathsf{hon}$, then property 4 follows from (T-REDIR). Properties 5 and 7 are trivial. Property 8 follows from (T-REDIR).
  - If $\iota_b = \mathsf{usr}$ and $\mu = \mathsf{att}$ then properties 3 and 6 are trivial. Property 4 follows from (T-REDIR). Property 8 follows from (T-REDIR).

- (T-REPLY): Property 1 follows directly from (T-REPLY) and property 8 is trivial. We perform a case distinction:
  - If $\iota_b \neq \mathsf{usr}$ Then we need to show property 2 which follows immediately from the typing rule, using Lemma 15
  - If $\iota_b = \mathsf{usr}$ and $\mu = \mathsf{hon}$, then properties 5, 7 and 4 of Definition 10 follow from (T-REPLY) and (T-FORM)
  - If $\iota_b = \mathsf{usr}$ and $\mu = \mathsf{att}$ then properties 6 and 4 of Definition 10 follow from (T-REPLY). Property 3 follows immediately from (T-REPLY) and from the observation that rule (T-BEREFFAIL) and (T-BASSIGNFAIL) are not used for typing the script, as the script can also be typed in the honest typing branch $b = \mathsf{hon}$.

- (T-REPLYERR): All claims are trivial.

□

We can now define the typing of websystems, which simply states that all browsers and servers contained in the system are well-typed.

Definition 13 (System Typing)**.** *Let W be a websystem. We write $\Gamma \vDash_{\ell_a,\mathsf{usr}} (\ell_a, \mathcal{K}) \rhd_{T_O} W$, if*

1. *for all $S \in servers(W)$ we have $\Gamma \vDash_{\ell_a,\mathsf{usr}} S$*
2. *for all $B \in browsers(W)$ we have $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$*
3. *for all $B \in browsers(W)$ with $B = (N, M, P, T, Q, \vec{a})^{\iota_b,l,\mu}$ and $N = \{n \mapsto u\}$ we have one of the following:*
   - *there exists $S \in servers(W)$ with $t \in running(S)$, $t = \lceil c \rfloor^{l',\mu}_{(n,u,\iota_b),(i,j)}$ and $l \sqsubseteq_I int_{\sqcup}(t)$ for some c, l', i, j,*
   - *or $I(\ell_a) \sqsubseteq_I \lambda(u)$*
   - *or $T_O = \{(\iota_b, n, u, l', \mu)\}$ for some l'*
4. *for all $v^\tau \in \mathcal{K}$ we have $\tau \sqsubseteq_{\ell_a} \ell_a$*

Next, we show that any script created by the attacker, that is served over a low integrity network connection is well-typed in the users browser.

Lemma 19 (Attacker Script Typability)**.** *For all scripts s and well formed environments Γ, URLs u with $I(\ell_a) \sqsubseteq_I I(\lambda(u))$, $\forall n^\tau \in values(s).\, \tau \sqsubseteq_{\ell_a} \ell_a$, $vars(s) = \emptyset$ we have $\Gamma, I(\ell_a), \mathsf{csrf} \vDash^{\mathsf{s}}_{\ell_a,\mathcal{P},u} s$.*

Proof. We first show that for all browser expressions be we have $\Gamma, \mathsf{csrf} \vDash^{\mathsf{be}}_{\ell_a} be : \ell_a$. We show the claim by induction over be.

- $be = r$.
  - if $C(\lambda(r)) \not\sqsubseteq_C C(\lambda(u))$ the claim follows immediately using rule (T-BEREFFAIL)
  - if $C(\lambda(r)) \sqsubseteq_C C(\lambda(u))$ we know by the well-formedness of Γ that $C(\mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))) \sqsubseteq_C C(\ell_a)$ and hence also $I(\ell_a) \sqsubseteq_I I(\mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r)))$ and the claim follows using (T-BEREF) and (T-BESUB)
- $be = v^\tau$: The claim follows from the assumption $\tau \sqsubseteq_{\ell_a} \ell_a$ and rule (T-BEVAL)
- $be = \mathsf{dom}(be', be'')$: Immediately by rule (T-BEDOM).
- $be = be_1 \odot be_2$: By induction and rule (T-BEBINOP)

We now show the main claim by induction over s.

- $s = s_1; s_2$: the claim follows from the induction hypothesis for s_1 and s_2 and (T-BSEQ)
- $s = \mathbf{skip}$: trivial with (T-BSKIP)
- $s = r := be$ We distinguish two cases
  - if $I(\lambda(u)) \not\sqsubseteq_I I(\lambda(r))$ then the claim is trivial with rule (T-BASSIGNFAIL).
  - if $I(\lambda(u)) \sqsubseteq_I I(\lambda(r))$ then we know because of $I(\ell_a) \sqsubseteq_I \lambda(u)$ that also $I(\ell_a) \sqsubseteq_I \lambda(r)$. Therefore, we know by well-formedness of Γ that if $\Gamma_{\mathcal{R}^@}(r) = \tau$ with $\tau = \mathtt{cred}(\cdot)$ then $C(\tau) \sqsubseteq_C C(\ell_a)$. We can hence show $\Gamma \vDash^{\mathsf{br}}_{\ell_a} r : \mathtt{ref}(\ell_a)$. Since we know that $\Gamma, b \vDash^{\mathsf{be}}_{\ell_a} be : \ell_a$ the claim follows.
- $s = \mathbf{setdom}(v, u, \vec{be})$: The claim follows from rule (T-BSETDOM), using our observation about expression types.
- $s = \mathbf{include}(u, \vec{be})$: The claim follows from rule (T-BINCLUDE), using our observation about expression types.

□

Next, we show that the attacker can only learn low confidentiality values from the network.

Lemma 20. *Attacker Knowledge for low confidentiality requests Whenever we have* $\alpha = \overline{\mathsf{req}}(\mathsf{usr}, n, u, p, ck, o)^{l,\mu}$ *with* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *and* $\lambda(u) \sqsubseteq_C C(\ell_a)$ *then for all* $n^\tau \in ns(p, ck)$ *we have* $C(\tau) \sqsubseteq_C C(\ell_a)$.

Proof. For $\mu = \mathsf{hon}$ the claim for $ns(p)$ follows immediately from the well-formedness of URLs and property 1 of Definition 9, otherwise the claim follows directly from property 2 of Definition 9,

The claim for $ns(ck)$ follows immediately from property 3 of Definition 9. □

The next two lemmas show that requests and responses crafted by the attacker are well-formed.

Lemma 21 (Attacker Request)**.** *Let* $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, o)^{\bot_I,\mathsf{att}}$ *with* $\iota_b \neq \mathsf{usr}$ *and for all* $v^\tau \in values(p, ck)$ *we have* $\tau \sqsubseteq_{\ell_a} \ell_a$. *Then* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$.

Proof. Since $\iota_b \neq \mathsf{usr}$ and $\mu = \mathsf{att}$, we have to show properties 2 and 4 of Definition 9. Both claims follow immediately since $\forall n^\tau \in ns(p, ck), \tau \sqsubseteq_{\ell_a} \ell_a$. □

Lemma 22 (Attacker Response)**.** *Let* $\alpha = \overline{\mathsf{res}}(\mathsf{usr}, n, u, u', \vec{v}, ck, page, s)^{\top_I,\mathsf{att}}$ *with* $I(\ell_a) \sqsubseteq_I \lambda(u)$ *and for all* $v^\tau \in values(p, ck)$ *we have* $\tau \sqsubseteq_{\ell_a} \ell_a$. *Then* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$

Proof. We show that α fulfills the properties of Definition 10.

We have to show properties 1, 6, 3, 4 and 8 of Definition 10.

With $\forall n^\tau \in ns(\vec{v}, ck, page, s), \tau \sqsubseteq_{\ell_a} \ell_a$. Properties 1 and 6 are trivial. Property 3 follows immediately from Lemma 19

For Property 4 we look at all $r \in ck$ and perform a case distinction:

- If $\lambda(u) \sqsubseteq_I \lambda(r)$ then by transitivity of $\sqsubseteq_I$ we know $I(\ell_a) \sqsubseteq_I \lambda(r)$ and hence by well-formedness of Γ we know that if $\Gamma_{\mathcal{R}^@}(r) = \mathsf{cred}(\cdot)$ then $C(\Gamma_{\mathcal{R}^@}(r)) \sqsubseteq_C C(\ell_a)$. We hence know that $\ell_a \sqsubseteq_{\ell_a} \mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$.
- If $\lambda(u) \not\sqsubseteq_I \lambda(r)$ then the property is trivially true.

For property 8, we have to show property 2 of Definition 9, which follows immediately. □

Finally, we show that whenever a well-formed system takes a step, it produces another well-typed system.

Lemma 23 (System Subject Reduction)**.** *Let* W *be a websystem with* $\Gamma \vDash_{\ell_a,\mathsf{usr}} (\ell_a, \mathcal{K}) \triangleright W$ *and* $(\ell_a, \mathcal{K}) \triangleright W \xrightarrow{\alpha} (\ell_a, \mathcal{K}') \triangleright W'$. *Then we have* $\Gamma \vDash_{\ell_a,\mathsf{usr}} (\ell_a, \mathcal{K}') \triangleright W'$

Proof. We do a proof by a case analysis over the derivation of $\xrightarrow{\alpha}$

- (A-NIL) If the step was taken using rule (A-NIL) then we perform an induction on the internal step. If the step is taken through rule (W-LPARALLEL) or (W-RPARALLEL) the claim follows by induction. If it is taken locally in one browser or server the claim follows from Lemma 6 or Lemma 14 and the fact that $\mathcal{K} = \mathcal{K}'$. Property 3 of Definition 13 follows from the observation that raising the server integrity label can only happen in rule (S-RESET).

- (A-BROSER) Follows immediately from Lemma 7 and Lemma 17. Property 3 follows from the semantics rules for browsers and servers. Property 4 follows from Lemma 20.

- (A-SERBRO) Follows immediately from Lemma 18 and Lemma 8. We can apply Lemma 8 because of property 3 of Definition 13.

- (A-TIMEOUTSEND) Follows immediately from Lemma 7.

- (A-TIMEOUTRECV) Let α' be the response sent in the rule. Then we trivially have $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha'$ and the claim follows using Lemma 8 and property Item 3 of Definition 13.

- (A-BROATK) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \overline{\mathsf{req}}(\mathsf{usr}, n, u, p, ck, o)^{l,\mu}$ and $I(\ell_a) \sqsubseteq_I \lambda(u)$. The typing of the browser follows immediately from Lemma 7. Property 4 follows from Lemma 20.

- (A-ATKSER) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{I(\ell_a),\mathsf{att}}$, where $ns(p, ck) \subset \mathcal{K}$.

  We hence get by Lemma 21 that $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ and the claim follows from Lemma 17.

- (A-SERATK) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \overline{\mathsf{res}}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l,\mu}$, where $n \in \mathcal{K}$.

  From Lemma 18 we get that the resulting server state is well-typed and that α is a well-formed response. We now have to show that for all $v^\tau \in \mathcal{K}'$ we have $\tau \sqsubseteq_{\ell_a} \ell_a$. Since $n \in \mathcal{K}$ and the only point where the attacker can learn n is in rule (A-ATKSER) we know that $\iota_b \neq \mathsf{usr}$ and $\mu = \mathsf{att}$. The claim follows directly from property 2 of Definition 10.

- (A-ATKBRO) We then have $W \xrightarrow{\alpha} W'$ with $\alpha = \mathsf{res}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l,\mu}$ where $\iota_b = \mathsf{usr}$, $l = \top_I$, $\mu = \mathsf{att}$ and $I(\ell_a) \sqsubseteq_I \lambda(u)$. By Lemma 22 we get that α is a well-formed response, then the claim follows from Lemma 8 (which we can apply, since $\mu = \mathsf{att}$ and $l = \top_I$).

□

B.2.5 Relation

We now define a notion of *High Equality* between different components, that will be used to relate two websystems.

The general intuition is, that everything that is of high integrity must be equal, while values of low integrity can be arbitrarily different.

Definition 14 (High Equality)**.** *We define high equality in different contexts:*

1. *For two (browser or server) expressions e, e' we inductively define $e =_{\perp_I} e'$ by the following rules.*

$$\frac{}{e =_{\perp_I} e} \qquad \frac{I(\ell_a) \sqsubseteq_I I(\tau) \sqcap_I I(\tau')}{v^{\tau} =_{\perp_I} v'^{\tau'}} \qquad \frac{I(\ell_a) \sqsubseteq_I I(\tau) \sqcap_I I(\tau')}{\mathit{fresh}()^{\tau} =_{\perp_I} \mathit{fresh}()^{\tau'}}$$

$$\frac{e_1 =_{\perp_I} e_2 \qquad e_2 =_{\perp_I} e_2'}{e_1 \odot e_1' =_{\perp_I} e_2 \odot e_2'} \qquad \frac{e_1 =_{\perp_I} e_1' \qquad e_2 =_{\perp_I} e_2'}{\mathsf{dom}(e_1, e_2) =_{\perp_I} \mathsf{dom}(e_1', e_2')}$$

$$\frac{|\vec{e}| = |\vec{e'}| \qquad \forall i \in [1 \ldots |\vec{e}|].e_i =_{\perp_I} e_i'}{\vec{e} =_{\perp_I} \vec{e'}}$$

2. *For two pages $page$, $page'$ we define $page =_{\perp_I} page'$ as*

$$\frac{\begin{array}{c} \mathit{dom}(page) = \mathit{dom}(page') \\ \forall v \in \mathit{dom}(page).\, (page(v) = \mathsf{form}(u_i, \vec{v_i})^{\mu} \wedge page'(v) = \mathsf{form}(u_i', \vec{v_i'})^{\mu'} \wedge \\ u_i =_{\perp_I} u_i' \wedge v_i =_{\perp_I} v_i') \end{array}}{page =_{\perp_I} page'}$$

3. *For two scripts s, s' we define $s =_{\perp_I} s'$ as*

$$\frac{}{\mathbf{skip} =_{\perp_I} \mathbf{skip}} \qquad \frac{s_1 =_{\perp_I} s_2 \qquad s_2 =_{\perp_I} s_2'}{s_1; s_1' =_{\perp_I} s_2; s_2'} \qquad \frac{be =_{\perp_I} be'}{r := be =_{\perp_I} r := be'}$$

$$\frac{\vec{be} =_{\perp_I} \vec{be'}}{\mathbf{include}(u, \vec{be}) =_{\perp_I} \mathbf{include}(u, \vec{be'})}$$

$$\frac{be =_{\perp_I} be' \qquad \vec{be} =_{\perp_I} \vec{be'}}{\mathbf{setdom}(be, u, \vec{be}) =_{\perp_I} \mathbf{setdom}(be', u, \vec{be'})}$$

4. *For two commands c, c' we define $c =_{\bot_I} c'$ as*

$$\frac{}{\mathbf{skip} =_{\bot_I} \mathbf{skip}} \qquad \frac{}{\mathsf{halt} =_{\bot_I} \mathsf{halt}} \qquad \frac{c_1 =_{\bot_I} c_2 \qquad c_2 =_{\bot_I} c_2'}{c_1; c_1' =_{\bot_I} c_2; c_2'}$$

$$\frac{se =_{\bot_I} se' \qquad c_1 =_{\bot_I} c_1' \qquad c_2 =_{\bot_I} c_2'}{\mathbf{if}\ se\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 =_{\bot_I} \mathbf{if}\ se'\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'}$$

$$\frac{se_1 =_{\bot_I} se_1' \qquad se_2 =_{\bot_I} se_2' \qquad se_3 =_{\bot_I} se_3'}{\mathbf{login}\ se_1, se_2, se_3 =_{\bot_I} \mathbf{if}\ se_1'\ \mathbf{then}\ se_2'\ \mathbf{else}\ se_3'} \qquad \frac{se =_{\bot_I} se'}{\mathbf{start}\ se =_{\bot_I} \mathbf{start}\ se'}$$

$$\frac{\forall i \in [1 \ldots |\vec{se}|].\, se_i =_{\bot_I} se_i'}{\mathbf{auth}\ \vec{se}\ \mathbf{at}\ l =_{\bot_I} \mathbf{auth}\ \vec{se'}\ \mathbf{at}\ l}$$

$$\frac{\forall i \in [1 \ldots |\vec{se}|].\, se_i =_{\bot_I} se_i'}{\mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se} =_{\bot_I} \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se'}}$$

$$\frac{\forall i \in [1 \ldots |\vec{se}|].\, se_i =_{\bot_I} se_i'}{\mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se} =_{\bot_I} \mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se'}}$$

$$\frac{\forall i \in [1 \ldots |\vec{se}|].\, se_i =_{\bot_I} se_i'}{\mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{se} =_{\bot_I} \mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{se'}}$$

5. *For two memories M, M' we write $M =_{\Gamma,\bot_I} M'$, if*

   - *for all $r \in dom(M) \cup dom(M')$ we have $M(r) =_{\bot_I} M'(r)$*
   - *for all $r \in dom(M) \setminus dom(M')$ with $M(r) = v^\tau$ we have $I(\ell_a) \sqsubseteq_I \tau$*
   - *for all $r \in dom(M') \setminus dom(M)$ with $M'(r) = v^\tau$ we have $I(\ell_a) \sqsubseteq_I \tau$*

6. *For two requests $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, o)^{l,\mu}$ (resp. $\alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{l,\mu}$) and $\beta = \overline{\mathsf{req}}(\iota_b', n', u', p', ck', o')^{l',\mu'}$ (resp. $\beta = \mathsf{req}(\iota_b', n', u', p', ck', o')^{l',\mu'}$) we let*

$$\begin{aligned} \alpha =_{\bot_I} \beta \iff & I(\ell_a) \not\sqsubseteq_I l \sqcup_I l' \Rightarrow \\ & \iota_b = \iota_b' \wedge n = n' \wedge u = u' \wedge \\ & dom(p) = dom(p') \wedge \forall x \in dom(p).\, p(x) =_{\bot_I} p'(x) \\ & ck =_{\Gamma,\bot_I} ck' \\ & \wedge\, l = l' \wedge \mu = \mu' \end{aligned}$$

7. *For two responses $\alpha = \overline{\mathsf{res}}(\iota_b, n, u, u_r, \vec{v}, ck, page, s)^{l,\mu}$ (resp. $\alpha = \mathsf{res}(\iota_b, n, u, u_r, \vec{v}, ck, page, s)^{l,\mu}$) and $\beta = \overline{\mathsf{res}}(\iota_b', n', u', u_r', \vec{v'}, ck', page', s')^{l,\mu'}$*

(resp. $\beta = \mathsf{res}(\iota_b', n', u', u_r', \vec{v'}, ck', page', s')^{\prime l,\mu'}$*) we let*

$$\begin{aligned}\alpha =_{\bot_I} \beta \iff & ck =_{\Gamma,\bot_I} ck' \\ & I(\ell_a) \not\sqsubseteq_I l \sqcup_I l' \Rightarrow \\ & \iota_b = \iota_b' \wedge n = n' \wedge u = u' \wedge u_r = u_r' \wedge \vec{v} =_{\bot_I} \vec{v'} \\ & page =_{\bot_I} page' \wedge s =_{\bot_I} s' \wedge \\ & l = l' \wedge \mu = \mu'\end{aligned}$$

8. *For two authentication events* $\alpha = \sharp[\vec{v}]_\ell^{\iota_b,\iota_u}$ *and* $\alpha' = \sharp[\vec{v'}]_{\ell'}^{\iota_b',\iota_u'}$ *we let* $\alpha =_{\bot_I} \alpha'$ *if*
   - $I(\ell_a) \sqsubseteq_I I(\ell)$ *and* $I(\ell_a) \sqsubseteq_I I(\ell')$ *or*
   - $\alpha = \alpha'$

We introduce a predicate $bad(\cdot)$ which we use to denote that the system has entered a state in which the browser will perform no more actions because it received an error message from the server.

Definition 15 (Bad State)**.**
- *A browser* $B = (N, K, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *is in a bad state and we write* $bad(B)$ *if* $\mathsf{halt} \in \vec{a}$*.*
- *A server* $S = (D, \phi, t)$ *is in a bad state and we write* $bad(S)$ *if there is a* $t \in running(S)$ *with* $t = \lceil c \rfloor_{E,R}^{l,\mu}$ *and*
  - $\iota_b = \mathsf{usr}$
  - $c = \mathbf{reply}\ (\mathsf{error}, \mathbf{skip}, \{\})$ *or* $c = \mathbf{bad}$
- *A web system* $(\ell_a, \mathcal{K}) \triangleright W$ *is in a bad state and we write* $bad(A)$ *if*
  - *with* $\{B\} = browsers(W)$*, we have* $bad(B)$
  - *for any* $S \in servers(W)$*, we have* $bad(S)$

The following properties are straightforward by inspecting the semantic rules:

- *If* $bad(A)$ *and* $A \xrightarrow{\vec{\alpha}}{}^* A'$*, then* $bad(A')$*.*
- *If* $bad(A)$ *and* $A \xrightarrow{\vec{\alpha}}{}^* A'$*, then there does not exist* $\sharp[\vec{v}]_\ell^{\iota_b,\iota_u} \in \vec{\alpha}$ *with* $I(\ell_a) \not\sqsubseteq_I I(\ell)$*.*

We now define a relation between two browsers.

Definition 16 (Browser Relation)**.** *Let* $B = (N, K, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *and* $B' = (N', K', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$ *be browsers. Then we write* $B \approx_\Gamma^B B'$ *if the following conditions hold*

1. $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B'$

2. $I(\ell_a) \sqsubseteq_I l \iff I(\ell_a) \sqsubseteq_I l'$ *and* $I(\ell_a) \not\sqsubseteq_I l \Rightarrow l = l'$
3. *If* $I(\ell_a) \not\sqsubseteq_I l$ *then* $N = N'$
4. $K =_{\Gamma_{\mathcal{R}^@}, \perp_I} K'$,
5. *For all* $t \in dom(P)$ *if* $P(t) = (u_1, page_1, l_1, \mu_1)$ *then* $I(\ell_a) \sqsubseteq_I l_1$ *or* $t \in dom(P')$ *with* $P'(t) = (u_2, page_2, l_2, \mu_2)$ *and* $u_1 = u_2$, $page_1 =_{\perp_I} page_2$ *and vice versa*
6. *If* $I(\ell_a) \not\sqsubseteq_I l$ *then* $dom(T) = dom(T')$ *and if* $T = \{t \mapsto s\}$ *and* $T' = \{t \mapsto s'\}$ *then* $s =_{\perp_I} s'$
7. $\vec{a} = \vec{a'}$
8. *If* $I(\ell_a) \not\sqsubseteq_I l$ *and* $Q = \{\alpha\}$ *then* $Q' = \{\alpha'\}$ *with* $\alpha =_{\perp_I} \alpha'$.

We let $B \approx^B_\Gamma B'$ *if*

- $bad(B)$
- *or* $B \approx^B_\Gamma B'$

Intuitively, two browsers are related by the relation $\approx^B_\Gamma$ if

1. Both browsers are well typed
2. Either both have low or high integrity. If the integrity is high, it must be the same.
3. If the integrity is high, then the network connections are equal
4. The cookie jars fulfill high equality
5. For any high integrity page in a tab of one browser, there exists a page in the same tab of the other browser, with same URL, integrity and attacked mode, and a DOM that fulfills high equality.
6. For high integrity browsers the scripts fulfill high equality
7. The list of user actions is equal
8. If the browsers are in high integrity states, then the events in the output buffer must fulfill high equality.

We then define the relation $\approx^B_\Gamma$, which holds if the left browser is in a bad state, or the browsers are in the relation $\approx^B_\Gamma$.

We then show that the relation $\approx^B_\Gamma$ is symmetric and transitive. Note that this does not hold for $\approx^B_\Gamma$.

Lemma 24 ($\approx^B_\Gamma$ is symmetric and transitive)**.** *The relation $\approx^B_\Gamma$ is symmetric and transitive.*

Proof. Trivial, by checking the individual properties of Definition 16 □

Next, we show that high equality on browser expressions is preserved under evaluation in the browser.

Lemma 25 (Preservation of $=_{\bot_I}$ under browser evaluation)**.** *Let be and be' be browser expressions with $be =_{\bot_I} be'$, let M, M' be memories with $M =_{\Gamma_{\mathcal{R}^@}, \bot_I} M'$, let u be a URL and let $page = f$ and $page' = f'$ be pages with $page =_{\bot_I} page'$. Then $eval_{\lambda(u)}(be, M, f) =_{\bot_I} eval_{\lambda(u)}(be', M', f')$.*

Proof. Let $v^\tau = eval_{\lambda(u)}(be, M, f)$ and $v'^{\tau'} = eval_{\lambda(u)}(be', M', f')$. If $I(\ell_a) \sqsubseteq_I I(\tau) \sqcap_I I(\tau')$ the claim is trivial. We hence now assume $I(\ell_a) \not\sqsubseteq_I I(\tau) \sqcap_I I(\tau')$, i.e., $I(\ell_a) \not\sqsubseteq_I I(\tau) \vee I(\ell_a) \not\sqsubseteq_I I(\tau')$

- $be = x$: Impossible, since evaluation is not defined on variables.
- $be = v^\tau$: Trivial, since evaluation on values is the identity ((BE-VAL)).
- $be = be_1 \odot be_2$: Then $be' = be'_1 \odot be'_2$ with $be_1 =_{\bot_I} be'_1$ and $be_2 =_{\bot_I} be'_2$. Let $v_1^{\tau_1} = eval_{\lambda(u)}(be_1, M, f)$, let $v_2^{\tau_2} = eval_{\lambda(u)}(be_2, M, f)$, let $v_1'^{\tau'_1} = eval_{\lambda(u)}(be'_1, M', f')$ and let $v_2'^{\tau'_2} = eval_{\lambda(u)}(be'_2, M', f')$. By induction we get $v_1^{\tau_1} =_{\bot_I} v_1'^{\tau'_1}$ and $v_2^{\tau_2} =_{\bot_I} v_2'^{\tau'_2}$. By rule (BE-BINOP) we know that $I(\tau) = I(\tau_1) \sqcup_I I(\tau_2)$ and $I(\tau') = I(\tau'_1) \sqcup_I I(\tau_2)'$.
  We know that $I(\ell_a) \not\sqsubseteq_I I(\tau)$ or $I(\ell_a) \not\sqsubseteq_I I(\tau')$. We perform a case distinction:
  - If $I(\ell_a) \not\sqsubseteq_I I(\tau)$ then we know that $I(\ell_a) \not\sqsubseteq_I I(\tau_1)$ and $I(\ell_a) \not\sqsubseteq_I It\tau_2$. By the definition of $=_{\bot_I}$ we then know that $v_1^{\tau_1} = v_1'^{\tau'_1}$ and $v_2^{\tau_2} = v_2'^{\tau'_2}$ and we get that $v^\tau = v'^{\tau'}$.
  - If $I(\ell_a) \not\sqsubseteq_I I(\tau')$ the claim follows analog.
- $be = r$: Then $be' = r$. By rule (BE-BE-REFERENCE) we have $v^\tau = M(r)$ and $v'^{\tau'} = M'(r)$ and the claim immediately follows because of $M =_{\bot_I} M'$.
- $be = \mathsf{dom}(be_1, be_2)$: Then $be' = \mathsf{dom}(be'_1, be'_2)$ with $be_1 =_{\bot_I} be'_1$ and $be_2 =_{\bot_I} be'_2$. Let $v_1^{\tau_1} = eval_{\lambda(u)}(be_1, M, f)$, let $v_2^{\tau_2} = eval_{\lambda(u)}(be_2, M, f)$, let $v_1'^{\tau'_1} = eval_{\lambda(u)}(be'_1, M', f')$ and let $v_2'^{\tau'_2} = eval_{\lambda(u)}(be'_2, M', f')$. By induction we get $v_1^{\tau_1} =_{\bot_I} v_1'^{\tau'_1}$ and $v_2^{\tau_2} =_{\bot_I} v_2'^{\tau'_2}$.
  We distinguish the following cases:
  - If $I(\ell_a) \sqsubseteq_I I(\tau_1) \sqcup_I I(\tau'_1)$ then by the definition of $=_{\bot_I}$ we also know that $I(\ell_a) \sqsubseteq_I I(\tau_2) \sqcup_I I(\tau'_2)$. Then the claim is trivial, since then $I(\ell_a) \sqsubseteq_I I(\tau) \sqcup_I I(\tau')$ by (BE-DOM).
  - If $I(\ell_a) \sqsubseteq_I I(\tau_2) \sqcup_I I(\tau'_2)$ the claim follows analog to the previous one.

– If $I(\ell_a) \not\sqsubseteq_I I(\tau_1)$, $I(\ell_a) \not\sqsubseteq_I I(\tau_1')$, $I(\ell_a) \not\sqsubseteq_I I(\tau_2)$ and $I(\ell_a) \not\sqsubseteq_I I(\tau_2')$ Then we know by that $v_1 = v_1'$ and $v_2 = v_2'$. The claim then follows from $page =_{\perp_I} page'$ and rule (BE-DOM).

□

Now we introduce the notion of deterministic termination. This property states that a system terminates and can only produce a single trace. This is a property that holds in the honest run, as the assumptions on user behaviour allow only terminating runs and without the attacker there is no point of non-determinism.

Definition 17 (Deterministic Termination)**.** *We say that a websystem W is deterministically terminating for a user* usr *if there exists exactly one unattacked trace γ such that $(\ell_a, \mathcal{K}) \triangleright W \xrightarrow{\gamma}{}^*(\ell_a, \mathcal{K}') \triangleright W'$ where $W' = B_{\mathsf{usr}}(M', P', \langle\rangle) \parallel W'$ and browsers$(W') = \emptyset$ for some $\mathcal{K}', W', M', P'$.*

We say that a server thread $t = \lceil c \rfloor_{E,R}^{l,\mu}$ is deterministically terminating if there exists exactly one $\vec{\alpha}$ with $t \xrightarrow{\alpha}{}^* t'$ for some $t' = \lceil c' \rfloor_{E',R'}^{l',\mu'}$ with

- $c' = \mathbf{reply}\ (page, \cdot, \cdot)\ \mathbf{with}\ \vec{x} = \cdot$, *where* $page \neq \mathsf{error}$
- *or* $c' = \mathbf{redirect}\ (\cdot, \cdot, \cdot)\ \mathbf{with}\ \vec{x} = \cdot$

We say that server S is deterministically terminating if all $t \in running(S)$ are deterministically terminating.

Note that it immediately follows that all servers in a deterministically terminating web system are also deterministically terminating.

Next we define a relation between two servers:

Definition 18 (Server Relation)**.** *Let $S = (D, \phi, t)$ and $S = (D', \phi', t')$ Then we write $S \approx_\Gamma^S S'$ if*

or the following conditions hold

1. $\Gamma \vDash_{\ell_a,\mathsf{usr}} S$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} S'$
2. *Let $t_1^H := \{t_1 | t_1 \in running(t) \wedge I(\ell_a) \not\sqsubseteq_I int_\sqcap(t_1)\}$ and $t_2^H := \{t_2 | t_2 \in running(t) \wedge I(\ell_a) \not\sqsubseteq_I int_\sqcap(t_2)\}$. Then there is a bijection $c : t_1^H \to t_2^H$ such that for all $t_1 \in t_1^H$ and $t_2 = c(t_1) \in t_2^H$, if we let $t_1 = \lceil c_1 \rfloor_{E_1,R_1}^{l_1,\mu_1}$ and $t_2 = \lceil c_2 \rfloor_{E_2,R_2}^{l_2,\mu_2}$ then we have*
   a) *$R_1 = R_1$ and with $E_1 = i_1, j_1$ and $E_2 = i_2, j_2$ we have $i_1 = i_2$ and $j_1 =_{\perp_I} j_2$.*
   b) *With $E_1 = (i_1, j_1)$ we have $D_{@1}(i_1) =_{\Gamma,\perp_I} D_{@2}(i_1)$*

c) the following holds:

i. $I(\ell_a) \sqsubseteq_I l_1 \iff I(\ell_a) \sqsubseteq_I l_2$ *and* $I(\ell_a) \not\sqsubseteq_I l_1 \Rightarrow l_1 = l_2$

ii. if $I(\ell_a) \not\sqsubseteq_I l_1$ *then* $c_1 =_{\perp_I} c_2$

iii. if $I(\ell_a) \sqsubseteq_I l_1$ *and there exists an* l *with* $I(\ell_a) \not\sqsubseteq_I l$ *and* c_1' *and* c_1'' *such that* $c_1 = c_1'; \mathsf{reset}\ l; c_1''$ *then there exist* c_2' *and* c_2'' *such that* $c_2 = c_2'; \mathsf{reset}\ l; c_2''$ *with* $c_1'' =_{\perp_I} c_2''$ *and vice versa.*

3. We have

- *for all* $j^\tau \in dom(D_\$) \cap dom(D_\$')$ *with* $C(\tau) \not\sqsubseteq_C C(\ell_a)$ *that* $D_\$(j) =_{\Gamma,\perp_I} D_\$'(j)$
- *for all* $j^\tau \in (dom(D_\$) \setminus dom(D_\$'))$ *with* $C(\tau) \not\sqsubseteq_C C(\ell_a)$ *that for all* $r \in \mathcal{R}$ *with* $I(\ell_a) \not\sqsubseteq_I \Gamma_{\mathcal{R}^@}(r)$ *we have* $D_\$(j)(r) = \perp$.
- *for all* $j^\tau \in (dom(D_\$') \setminus dom(D_\$))$ *with* $C(\tau) \not\sqsubseteq_C C(\ell_a)$ *that for all* $r \in \mathcal{R}$ *with* $I(\ell_a) \not\sqsubseteq_I \Gamma_{\mathcal{R}^@}(r)$ *we have* $D_\$'(j)(r) = \perp$.

4. For all j^τ *with* $I(\ell_a) \not\sqsubseteq_I \tau$ *we have that* $\phi(j) = \phi'(j)$.

We let $S \approx_\Gamma^S S'$ *if*

- $bad(S)$
- *or* $S \approx_\Gamma^S S'$ *and* S' *is deterministically terminating.*

Intuitively, two servers are in the relation $\approx_\Gamma^S$ if

1. Both servers are well-typed.
2. There is a bijection between high integrity running threads on the two servers. For each pair we have that
   a) They have the same request context and global memory index. For high integrity threads they also have the same session memory index.
   b) High equality holds between the two global memories.
   c) i. Either both threads have high or both have low integrity. If it is high it has to be equal.
      ii. For high integrity threads, the two codes have to be high equal
      iii. If the integrity of one thread is low, but it can be raised to high using a reset command, then there also has to be a reset with the same high integrity label in the other thread.
3. • For all session identifiers appearing in both threads that are secret, the session memories indexed by the identifiers are high equal

- For all session identifiers present in only one thread, that are secret, all high integrity references are unset.

4. For all high integrity session identifiers, the user information (ϕ) is equal.

We then define the relation $\approx^S_\Gamma$ which holds if the left server is in a bad state, or the servers are in the relation $\approx^S_\Gamma$ and the right server is deterministically terminating.

We then show that the relation $\approx^S_\Gamma$ is symmetric and transitive. Note that this does not hold for $\approx^S_\Gamma$.

Lemma 26 ($\approx^S_\Gamma$ is symmetric and transitive)**.** *The relation $\approx^S_\Gamma$ is symmetric and transitive.*

Proof. Trivial, by checking the individual properties of Definition 16 □

Next, we show that high equality for server expressions is preserved under evaluation.

Lemma 27 (Preservation of $=_{\bot_I}$ under server evaluation)**.** *Let se and se' be server expressions with $se =_{\bot_I} se'$, let $E = i, j$ and $E' = i, j'$ with $j =_{\bot_I} j'$, let D, D' be databases and $\Gamma_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}$ typing environments with $D_@(i) =_{\Gamma_{\mathcal{R}^@}, \bot_I} D_@(i')$ and if $j \neq \bot$ then $D_\$(j) =_{\Gamma'_{\mathcal{R}^\$}, \bot_I} D'_\$(j')$ with $\forall r. \Gamma'_{\mathcal{R}^\$}(r) = \Gamma_{\mathcal{R}^\$}(r) \sqcap jlabel(j)$. Then $eval_E(se, D) =_{\bot_I} eval_{E'}(se', D')$.*

Proof. Let $v^\tau = eval_E(se, D)$ and $v'^{\tau'} = eval_{E'}(se', D')$. If $I(\ell_a) \sqsubseteq_I I(\tau) \sqcap_I I(\tau')$ the claim is trivial. We hence now assume $I(\ell_a) \not\sqsubseteq_I I(\tau) \sqcap_I I(\tau')$, i.e., $I(\ell_a) \not\sqsubseteq_I I(\tau) \vee I(\ell_a) \not\sqsubseteq_I I(\tau')$

- $be = x$: Impossible, since evaluation is not defined on variables.
- $be = v^\tau$: Trivial, since evaluation on values is the identity (E-VAL)
- $be = \mathit{fresh}()^\tau$: Then $be' = \mathit{fresh}()^\tau$. By rule (SE-FRESH) we know that $v, v' \in \mathcal{N}_\tau$. For simplicity, we assume that $v = v'$ and that the sampled names are fresh (i.e., have not been sampled before and will not be sampled again).
- $se = se_1 \odot se_2$: Then $se' = se'_1 \odot se'_2$ with $se_1 =_{\bot_I} se'_1$ and $se_2 =_{\bot_I} se'_2$ and the claim follows by induction analog to Lemma 25.
- $se = @r$: Then $se' = @r$. The claim then follows from $D_@(i) =_{\Gamma_{\mathcal{R}^@}, \bot_I} D'_@(i')'$.
- $se = \$r$: Then $se' = \$r$ and the claim then follows from $D_\$(j) =_{\Gamma_{\mathcal{R}^\$}, \bot_I} D'_\$(j')$.

□

Now we introduce a relation between websystems:

Definition 19 (Integrity Relation)**.** *Given a typing environment Γ, we consider two websystems $A = (\ell_a, \mathcal{K}) \triangleright W$ and $A' = (\ell_a, \mathcal{K}') \triangleright W'$ to be in the integrity relation $\approx_\Gamma$ if*

1. $\Gamma \vDash_{\ell_a,\mathsf{usr}} W$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} W'$
2. *For each server* $S \in$ *servers*(W) *there exists exactly one server* $S' \in$ *servers*$(W)'$ *such that urls*$(S) =$ *urls*(S') *and vice versa, i.e. the available URLs and the code associated to them are the same in both web systems. We will call these servers* S *and* S' corresponding *servers. Formally, the correspondence is a bijection between the sets servers*(W) *and servers*(W').
3. *For all servers* S *in* W *and the corresponding servers* S' *in* W' *we have that* $S \approx^S_\Gamma S'$
4. W *contains exactly one browser* $B = (N, K, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$, *and* W' *contains exactly one browser* $B' = (N', K', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$. *and we have* $B \approx^B_\Gamma B'$.

We furthermore let $A \cong_\Gamma A'$ *if*

1. $bad(A)$
2. *or* $A \approx_\Gamma A'$ *and* A' *is deterministically terminating.*

Intuitively, we require that

1. Both websystems are well-typed
2. All servers have a matching server in the other websystem that contains the same URLS and commands (i.e., statically the websystems are equal)
3. All corresponding servers are in the relation $\approx^S_\Gamma$.
4. Both websystems contain exactly one browser, and they are in the relation $\approx^B_\Gamma$

We then define the relation $\cong_\Gamma$, which holds if the left system is in a bad state, or the systems are in the relation $\approx_\Gamma$ and the right system is deterministically terminating.

Next, we show that the relation $\approx_\Gamma$ is transitive. This property is helpful for proofs of upcoming lemmas, where we consider the case where only one of the system does a step. Then it is enough to show that the system before and after taking the step are in the relation.

Lemma 28 (Transitivity of $\cong_\Gamma$)**.** *The relation* $\cong_\Gamma$ *is transitive.*

Proof. Trivial, by inspecting the single conditions. □

Now we show that whenever a browser processes an event with low sync integrity for an internal step, then the state before and after taking the step are in the relation.

Lemma 29 (Low Sync Integrity Browser Steps)**.** *Let* $B = (N, K, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *and* $B' = (N', K', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$ *be browsers with* $B \xrightarrow{\bullet@l''} B'$ *and* $I(\ell_a) \sqsubseteq_I l''$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$. *Then* $B \approx^B_\Gamma B'$

Proof. We show that all the properties of Definition 16 are fulfilled. In all cases property 1 follows immediately from Lemma 6. Proof by induction over the derivation of the step α

- (B-SEQ) follows from induction.
- (B-SKIP) Properties 2, 5, 4, 3, 7 and 8 are trivial, since $l = l'$, $P = P'$, $K = K'$, $N = N'$, $\vec{a} = \vec{a'}$ and $Q = Q'$. Property 6 is trivial, since because of $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l$.
- (B-END) Impossible, since $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$.n
- (B-SETREFERENCE) Properties 2, 5, 3, 7 and 8 are trivial, since $l = l'$, $P = P'$, $N = N'$, $\vec{a} = \vec{a'}$ and $Q = Q'$. Property 6 is trivial, since because $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l$.

  We know that $T = \{t \mapsto r := be\}$.

  For property 4, by rule (T-BASSIGN) we then know that $I(\ell_a) \sqsubseteq_I \text{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$ and by property 2 of Definition 11 we then know that with $K(r) = v^\tau$ we have $\tau = \text{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$. With $K'(r) = v'^{\tau'}$ we also have $\tau' = \text{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$. Because of $I(\ell_a) \sqsubseteq_I \text{ref}_\tau(\Gamma_{\mathcal{R}^@}(r))$ the claim follows immediately.
- (B-SETDOM) Properties 2, 4, 3, 7 and 8 are trivial, since $l = l'$, $K = K'$, $N = N'$, $\vec{a} = \vec{a'}$ and $Q = Q'$. Property 6 is trivial, since because $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l$.

  We know that $T = \{tab \mapsto \mathbf{setdom}(v, u, \vec{be})\}$ and $P(tab) = (u', f, l_P, \mu_P)$

  Since by property 4 we know $l_P = l$ we have $I(\ell_a) \sqsubseteq_I l_P$ and the claim is trivial.
- (B-LOAD): Impossible since $I(\ell_a) \sqsubseteq_I l$
- (B-SUBMIT): Impossible since $I(\ell_a) \sqsubseteq_I l$
- (B-INCLUDE): Properties 2, 5, 4 and 7 are trivial, since $l = l'$, $P = P'$, $K = K'$ and $\vec{a} = \vec{a'}$. Properties 3, 6 and 8 are trivial, since because $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l \sqcup_I l'$.

□

We show the same for internal steps on the server side with low sync integrity.

Lemma 30 (Low Sync Integrity Server Steps)**.** *Let $S = (D, \phi, t)$ and $S = (D', \phi', t')$ with $S \xrightarrow{\alpha} S'$ and $\alpha \in \{\bullet, \sharp[\cdot]:, \mathsf{error}\}$ and $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$ and $\Gamma \vDash_{\ell_a,\mathsf{usr}} S$. Then $S \approx^S_\Gamma S'$*

Proof. We show that all the properties of Definition 18 are fulfilled. In all cases property 1 follows immediately from Lemma 14.

Let $t_1 \in \text{running}(S)$ and let $t_1 = \lceil c \rfloor^{l,\mu}_{E,R}$. Then there exists $t'_1 \in \text{running}(S')$ with $t'_1 = \lceil c' \rfloor^{l',\mu'}_{E',R}$ and $(D, \phi, t_1) \xrightarrow{\alpha} (D', \phi', t'_1)$

We prove the claim by induction over the derivation of step α.

- (S-SEQ) The claim follows from the induction hypothesis
- (S-IFTRUE) Properties 2a, 2b 3 and 4 are trivial since $I(\ell_a) \sqsubseteq_I l$, $E = E'$, $D = D'$ and $\phi = \phi$. Property 2c is trivial since $I(\ell_a) \sqsubseteq_I l$ and $I(\ell_a) \sqsubseteq_I l'$
- (S-TCTRUE) All properties are trivial since since $I(\ell_a) \sqsubseteq_I l$, $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.
- (S-SKIP) All properties are trivial since $I(\ell_a) \sqsubseteq_I l$, $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.
- (S-RESET) Then $c = \text{reset } l''$ Properties 2a, 2b 3 and 4 are trivial since $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$. Property 2c is trivial since $I(\ell_a) \sqsubseteq_I l$ (because the reset command will never lower the integrity label) and $I(\ell_a) \sqsubseteq_I l'$.
- (S-IFFALSE) Analog to rule (S-IFTRUE)
- (S-TCFALSE) All properties are trivial since since $I(\ell_a) \sqsubseteq_I l$, $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$. $Property(iii)$ of 2c does hold since we know that a token check is never followed by a reset – this is enforced in (S-IFFALSE) and (S-IFTRUE)
- (S-RESTORESESSION) Properties 2b, 2c 3 and 4 are trivial since $D = D'$, $l = l'$ and $\phi = \phi$. Let $E = i, j$ and $E' = i, j'$. If for t_1 we do not have $I(\ell_a) \not\sqsubseteq_I int_\sqcap(t_1)$, property 2a is trivial. Otherwise, we know that $\mu = \text{hon}$. Then we know by rule (T-RUNNING) and (T-START) that because of $I(\ell_a) \sqsubseteq_I l$ we have $I(\ell_a) \sqsubseteq_I I(jlabel(j))$ and $I(\ell_a) \sqsubseteq_I I(jlabel(j'))$. Property 2a immediately follows.
- (S-NEWSESSION) Properties 2b, 2c and 4 are trivial since with $D = (D_@, D_\$)$, $D' = (D'_@, D'_\$)$ we have $D_@ = D'_@$, $l = l'$ and $\phi = \phi$. If for t_1 we do not have $I(\ell_a) \not\sqsubseteq_I int_\sqcap(t_1)$, property 2a is trivial. Otherwise, we know that $\mu = \text{hon}$. Then we know by rule (T-RUNNING) and (T-START) that because of $I(\ell_a) \sqsubseteq_I l$ we have $I(\ell_a) \sqsubseteq_I I(jlabel(j))$ and $I(\ell_a) \sqsubseteq_I I(jlabel(j'))$. Property 2a immediately follows.

  Let $E' = i, j'$ For property 3 we know that $j' \in D'_\$ \setminus D_\$$. Since for all r, $(D'_\$(j')(r) = \bot$ ($D'_\$(j')$ is a fresh memory) property 3 immediately follows.
- (S-SETGLOBAL) Then we have $c = @r := se$.

  Properties 2a, 2c 3 and 4 are trivial since with $D = (D_@, D_\$)$, $D' = (D'_@, D'_\$)$ we have $E = E'$, $D_\$ = D'_\$$, $l = l'$ and $\phi = \phi$.

  Let $E = i, j$ and let $v^\tau = D_@(i)(r)$ and $v'^{\tau'} = D'_@(i)(r)$. By rule (T-SETGLOBAL) we know that $I(\ell_a) \sqsubseteq_I I(\text{ref}_\tau(\Gamma_{\mathcal{R}^@}(r)))$. By property 1 we know that $\tau = \text{ref}_\tau(\Gamma_{\mathcal{R}^@}(r)) = \tau'$. We hence have $I(\ell_a) \sqsubseteq_I I(\tau)$ and $I(\ell_a) \sqsubseteq_I I(\tau')$ and the claim follows immediately.
- (S-SETSESSION) Then we have $c = \$r := se$.

  Properties 2a, 2c 2b and 4 are trivial since with $D = (D_@, D_\$)$, $D' = (D'_@, D'_\$)$ we have $E = E'$, $D_@ = D'_@$, $l = l'$ and $\phi = \phi$.

Let $E = i, j$ and let $v^{\tau} = D_{\$}(j)(r)$ and $v'^{\tau'} = D'_{\$}(j)(r)$. By rule (T-SETSESSION) we know that $I(\ell_a) \sqsubseteq_I I(\text{ref}_{\tau}(\Gamma_{\mathcal{R}^{\$}}(r)) \tilde{\sqcup}_I jlabel(j))$. By property 1 we know that $\tau = \text{ref}_{\tau}(\Gamma_{\mathcal{R}^{@}}(r)) \tilde{\sqcup}_I jlabel(j) = \tau'$. We hence have $I(\ell_a) \sqsubseteq_I I(\tau)$ and $I(\ell_a) \sqsubseteq_I I(\tau')$ and the claim follows immediately.

- (S-LOGIN) $t_1 = \lceil \textbf{login}\ se_{usr}, se_{pw}, se_{side} \rfloor^{l,\mu}_{E,R}$.

  Properties 2a, 2b 2c and 3 are trivial since $E = E'$, $D = D'$ and $l = l'$.

  By rule (T-LOGIN) with $\Gamma, \ell_s \vDash^{\mathsf{se}}_{\ell_a} se_{sid} : \tau$ we get that $I(\ell_a) \sqsubseteq_I \tau$. Property 4 then follows immediately using Lemma 5.

- (S-AUTH) Properties 2a, 2b 2c 3 and 4 are trivial since $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.

- (S-OCHKSUCC) All properties are trivial since since $I(\ell_a) \sqsubseteq_I l$, $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.

- (S-OCHKFAIL) All properties are trivial since since $I(\ell_a) \sqsubseteq_I l$, $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$.

- (S-LPARALLEL) The claim follows from induction hypothesis

- (S-RPARALLEL) The claim follows from induction hypothesis

□

Now, we show the same for browsers issuing a request with low sync integrity.

Lemma 31 (Low Sync Integrity Browser Request)**.** *Let* $B = (N, K, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *and* $B' = (N', K', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$ *be browsers with* $B \xrightarrow{\alpha} B'$ *and* $I(\ell_a) \sqsubseteq_I sync_I(\alpha)$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$ *and* $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, o, ck)^{l'',\mu''}$ *Then* $B \approx^B_{\Gamma} B'$ *and* $sync_I(\alpha) \sqsubseteq_I l''$.

Proof. We show that all the properties of Definition 16 are fulfilled. We know that α has been produces using rule (B-FLUSH) Property 1 follows immediately from Lemma 7.

Properties 2, 5, 4, 3, 6 and 7 are trivial, since $l = l'$, $P = P'$, $K = K'$, $N = N'$, $T = T'$ and $\vec{a} = \vec{a'}$. Properties 8 is trivial, since because $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l \sqcup_I l'$.

The claim $\text{sync}_I(\alpha) \sqsubseteq_I l''$ follows immediately, by inspecting the rules (B-LOAD), (B-INCLUDE), (B-SUBMIT) and (B-REDIRECT). □

Next, we show the same for a browser receiving a response of low sync integrity .

Lemma 32 (Low Sync Integrity Browser Response)**.** *Let* $B = (N, K, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *and* $B' = (N', K', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$ *be browsers with* $B \xrightarrow{\alpha} B'$ *and* $I(\ell_a) \sqsubseteq_I sync_I(\alpha)$ *and* $\alpha = \mathsf{res}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l'',\mu''}$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$. *Then* $B \approx^B_{\Gamma} B'$ *and* $sync_I(\alpha) \sqsubseteq_I l''$.

Proof. We show that all the properties of Definition 16 are fulfilled. We perform a case distinction on the rule used to derive α.

In all cases for property 4 we get from property 4 of 10 that for all updated references r, we have $I(\ell_a) \sqsubseteq_I I(\mathrm{ref}_\tau(\Gamma_{\mathcal{R}^@}))$. The claim then follows from property 2 for B and B'

- (B-RECVLOAD): The claim $\mathrm{sync}_I(\alpha) \sqsubseteq_I l''$ follows from the observation that the integrity can only be lowered between the request and the response

  Property 1 follows immediately from Lemma 8.

  Property 7 is trivial, since we have $\vec{a} = \vec{a'}$.

  Properties 3, 2 6, 8 are trivial, since because $I(\ell_a) \sqsubseteq_I \mathrm{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l$ and $I(\ell_a) \sqsubseteq_I l'$.

  Property 5 follows immediately from $I(\ell_a) \sqsubseteq_I l''$, which we get from $\mathrm{sync}_I(\alpha) \sqsubseteq_I l''$.

- (B-RECVINCLUDE) The claim $\mathrm{sync}_I(\alpha) \sqsubseteq_I l''$ is trivial.

  Property 1 follows immediately from Lemma 8.

  Properties 5 and 7 are trivial, since we have $P = P'$ and $\vec{a} = \vec{a'}$.

  Properties 3, 2 6, 8 are trivial, since because $I(\ell_a) \sqsubseteq_I \mathrm{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l$ and $I(\ell_a) \sqsubseteq_I l'$.

- (B-REDIR) The claim $\mathrm{sync}_I(\alpha) \sqsubseteq_I l''$ is trivial.

  Property 1 follows immediately from Lemma 8.

  Properties 5 and 7 are trivial, since we have $P = P'$ and $\vec{a} = \vec{a'}$.

  Properties 2, 3, 6 and 8 are trivial, since because $I(\ell_a) \sqsubseteq_I \mathrm{sync}_I(\alpha)$ we know $I(\ell_a) \sqsubseteq_I l'$ and $I(\ell_a) \sqsubseteq_I l$.

□

Now we show the same for servers receiving a request of low sync integrity.

Lemma 33 (Low Sync Integrity Server Request)**.** *Let $S = (D, \phi, t)$ and $S = (D', \phi', t')$ with $S \xrightarrow{\alpha} S'$ and $I(\ell_a) \sqsubseteq_I sync_I(\alpha)$ and $\alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{l'', \mu''}$ $\Gamma \vDash_{\ell_a, \mathsf{usr}} S$ and $\Gamma \vDash_{\ell_a, \mathsf{usr}} \alpha$. Then $S \approx^S_\Gamma S'$*

Proof. We show that all the properties of Definition 18 are fulfilled. Properties 2a, 2b 2c are trivial since $I(\ell_a) \sqsubseteq_I \mathrm{sync}_I(\alpha)$. 3 and 4 are trivial since with $D = (D_@, D_\$)$, $D' = (D'_@, D'_\$)$ we have $D_\$ = D'_\$$, and $\phi = \phi$. Property 1 follows immediately from Lemma 17. □

Now we show the same for servers sending a response of low sync integrity.

Lemma 34 (Low Sync Integrity Server Response)**.** *Let* $S = (D, \phi, t)$ *and* $S = (D', \phi', t')$ *with* $S \xrightarrow{\alpha} S'$ *and* $I(\ell_a) \sqsubseteq_I sync_I(\alpha)$ *and* $\alpha = \overline{\mathsf{res}}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l'', \mu''}$ *and* $\Gamma \vDash_{\ell_a, \mathsf{usr}} S$*. Then* $S \approx^S_\Gamma S'$

Proof. Then the event α was produced using rule (S-REPLY) or (S-REDIR). In both cases properties 2a, 2b 2c 3 and 4 are trivial since $E = E'$, $D = D'$, $l = l'$ and $\phi = \phi$. Property 1 follows immediately from Lemma 18. □

Finally, we use the previous lemmas to show that if a websystem takes a step of low sync integrity, then the state before and after the step are in the relation.

Lemma 35 (Low Sync Integrity Steps)**.** *Let* A, A' *be web systems with* $A \xrightarrow{\alpha} A'$ *for some* α *with* $I(\ell_a) \sqsubseteq_I sync_I(\alpha)$*. Then* $A \approx_\Gamma A'$*.*

Proof. We perform an induction on the rule used to derive the step α.

- (A-NIL) Then, if the step is derived through rule (W-LPARALLEL) or (W-RPARALLEL) the claim follows by induction. The claim for internal browser steps follows from Lemma 29. The claim for internal server steps follows from Lemma 30.
- (A-BROWSERSERVER) Then we know by Lemma 31 that the browser relation is preserved and that the server step is of low integrity. By Lemma 7 we know that the request is well typed. The claim for the server relation then follows from Lemma 33.
- (A-SERVERBROWSER) Then by Lemma 18 we get that the response is well-typed. By Lemma 31 we hence know that the browser relation is preserved and that the server step is of low integrity. Then we know by Lemma 34 that the server relation is preserved.
- (A-TIMEOUTSEND) Then the claim follows from Lemma 31.
- (A-TIMEOUTRECV) Then the claim follows from Lemma 32.
- (A-BROATK) Then the claim follows from Lemma 31 for the browser step.
- (A-ATKSER) Then the claim follows from Lemma 33 for the server step, using Lemma 21
- (A-SERATK) Then the claim follows from Lemma 34 for the server step.
- (A-ATKBRO) Then the claim follows from Lemma 32 for the browser step, using Lemma 22

□

We now define the *next high integrity state* of a deterministically terminating websystem as the state that is just before processing the next event with high sync integrity. This state can be reached by processing a number of events with low sync integrity. We furthermore show that

1. this state is unique
2. The websystem before and after taking the steps with low sync integrity are in the relation.
3. The websystem in the newly reached state is still deterministically terminating
4. The websystem has a special form (one of the few specified in the lemma)

Lemma 36 (Low Integrity Catch Up)**.** *Let $A = (\mathcal{K}, \ell_a) \triangleright W$ be a deterministically terminating websystem. We say that it is in a low integrity state if:*

- $\{B\} = \mathit{browsers}(W)$, $B = (N, M, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *and* $I(\ell_a) \sqsubseteq_I l$
- *or there is a* $t \in \{t' \mid S \in \mathit{servers}(W) \wedge t' \in \mathit{running}(S)\}$ *with* $t = \lceil c \rfloor^{l,\mu}_{R,E}$ *with*
  - $\mathbf{halt} \notin \mathit{coms}(c)$
  - $I(\ell_a) \sqsubseteq_I l$
  - $R = n, u, \iota_b, o \wedge \iota_b = \mathsf{usr}$

We then let $\mathbf{nexth}(A)$ *be the websystem* $A' = (\mathcal{K}', \ell_a) \triangleright W'$ *such that*

- $A \xrightarrow{\beta}{}^* A'$ *with* $I(\ell_a) \sqsubseteq_I \mathit{sync}_I(\beta)$ *for all* $\beta \in \vec{\beta}$.
- *for all* A'', α *with* $A' \xrightarrow{\alpha} A''$ $I(\ell_a) \not\sqsubseteq_I \mathit{sync}_I(\alpha)$

We then know that

1. *There exists such a unique* A'
2. $A \approx_\Gamma A'$
3. A' *is deterministically terminating*
4. *Let* $B = (N, M, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *be the honest browser in* W *and let* B' *be the honest browser in* W'*. Then exactly one of the following claims about* W' *holds*
   a) $B' = (\{\}, M', P', \{tab \mapsto \mathbf{skip}\}, \{\}, \vec{a'})^{\mathsf{usr},l',\mu'}$
   b) *there exists a server* S *in* W' *with* $t \in \mathit{running}(S)$ *and* $t = \lceil \mathsf{reset}\ l''; c \rfloor^{l',\mu'}_{R,E}$ *and* $I(\ell_a) \not\sqsubseteq_I l''$
   c) $B' = (N', M', P', T', \{\}, \vec{a'})^{\mathsf{usr},l',\mu'}$ *with* $N = \{n \mapsto \_\}$, $I(\ell_a) \not\sqsubseteq_I l'$ *and there exists a server* S *in* W' *with* $t \in \mathit{running}(S)$ *and* $t = \lceil \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{se} \rfloor^{l',\mu'}_{R,E}$ *and* $R = n, \_, \_, \_$.
   d) $B' = (N', M', P', T', \{\}, \vec{a'})^{\mathsf{usr},l',\mu'}$ *with* $N = \{n \mapsto \_\}$, $I(\ell_a) \not\sqsubseteq_I l'$ *and there exists a server* S *in* W' *with* $t \in \mathit{running}(S)$ *and* $t = \lceil \mathbf{redirect}\ (u, p, ck)\ \mathbf{with}\ \vec{se} \rfloor^{l',\mu'}_{R,E}$ *and* $R = n, \_, \_, \_$.

e) $B' = (N', M', P', T', \{\}, \vec{a'})^{\mathsf{usr},l',\mu'}$ *with* $N = \{n \mapsto \_\}$, $I(\ell_a) \not\sqsubseteq_I l'$ *and* $T'_O = \{(\_, n, \_, \_, \_)\}$

Proof. We show that the different claims hold:

1. The existence and uniqueness follow immediately from the fact the W is deterministically terminating, using Definition 17
2. $A \approx_\Gamma A'$ follows from repeated application of Lemma 35 and Lemma 28.
3. Deterministic termination for W'follows immediately from deterministic termination of W, using Definition 17
4. The form of W' follows from the observation that these five points are the only ones in the semantic rules, where the integrity is raised.

□

Next, we show that if two high integrity browsers are in the relation and the left browser takes an internal step of high sync integrity, then also the right browser can take the same step and the resulting browsers are still in the relation.

Lemma 37 (High Sync Integrity Browser Steps)**.** *Let* $B_1 = (N, M, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *and* $B_2 = (N', M', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$ *be browsers with* $B_1 \approx^B_\Gamma B_2$ *and* $I(\ell_a) \not\sqsubseteq_I l$ *and let* $B_1 \xrightarrow{\bullet@l_s} B'_1$ *with* $I(\ell_a) \not\sqsubseteq_I l_s$*. Then there exist* B'_2 *such that* $B_2 \xrightarrow{\bullet@l_s} B'_2$ *and* $B'_1 \approx^B_\Gamma B'_2$*.*

Proof. We show that all properties of Definition 16 are fulfilled. In all cases property 6 follows immediately from Lemma 6. Because of $B_1 \approx^B_\Gamma B_2$ we know

- $\Gamma \vDash_{\ell_a,\mathsf{usr}} B$ and $\Gamma \vDash_{\ell_a,\mathsf{usr}} B'$
- $l = l'$
- $N = N'$
- $K =_{\Gamma_{\mathcal{R}^@},\bot_I} K'$
- $dom(T) = dom(T')$ and if $T = \{t \mapsto s\}$ and $T' = \{t \mapsto s'\}$ then $s =_{\bot_I} s'$
- $\vec{a} = \vec{a'}$

By property 1 of Definition 11 and because of $I(\ell_a) \not\sqsubseteq_I l$ we know that $\mu = \mathsf{hon}$.

We perform an induction on the derivation of the step α.

- (B-SEQ) The claim follows by induction.

- (B-SKIP) Trivial because $s =_{\bot_I} s'$
- (B-END) Trivial because $s =_{\bot_I} s'$.
- (B-SETREFERENCE) Then because of $N = N'$, $s =_{\Gamma,\bot_I} s'$, we can also apply (B-SETREFERENCE) for B_2. We have that $s = r := be$ and $s' = r := be'$ where $be =_{\bot_I} be'$ and the claim follows immediately using Lemma 25.
- (B-SETDOM) Then because of $N = N'$, $s =_{\Gamma,\bot_I} s'$, we can also apply (B-SETDOM) for B_2. We have that $s = \mathbf{setdom}(be, u, \vec{be})$ and $s' = \mathbf{setdom}(be', u, \vec{be'})$ where $be =_{\bot_I} be'$ and $\forall k \in [1 \dots |\vec{be}|].\, be_k =_{\bot_I} be'_k$. By rule (T-BSETDOM) we known that $be = v^{\tau}$ and $be' = v'^{\tau'}$ are primitive values with $I(\tau) = I(\tau') = \bot_I$. Hence we know $v = v'$ by the definition of $=_{\bot_I}$. Using Lemma 25 for all expressions in $\vec{be}$, we get $page =_{\bot_I} page'$ and the claim follows.
- (B-LOAD) Because of $N = N'$, $dom(T) = dom(T')$, and $\vec{a} = \vec{a'}$ we can also apply rule (B-LOAD) in B_2

  All properties except for property 3 and 8 are trivial.

  For simplicity we assume that the names n and n' sampled in the two browser are the same, i.e., we have $n = n'$, and property 3 follows immediately,

  For property 8 the only non-trivial condition is the claim on the cookies of the produced event. This however follows immediately from $K =_{\Gamma_{\mathcal{R}^@},\bot_I} K'$
- (B-INCLUDE) Because of $N = N'$, $s =_{\Gamma,\bot_I} s'$, we can also apply (B-INCLUDE) for B_2. For simplicity we assume that the names n and n' sampled in the two browser are the same, i.e., we have $n = n'$, and property 3 follows immediately using property 4 to get that the DOM is of high integrity and hence the origins of the two requests are the same. For property 8 the only non-trivial conditions are the claim on the parameters and the cookies of the produced event. These however follow immediately from $s =_{\bot_I} s'$ using Lemma 25 and $K =_{\Gamma_{\mathcal{R}^@},\bot_I} K'$.
- (B-SUBMIT) Because of $N = N'$, $dom(T) = dom(T')$, and $\vec{a} = \vec{a'}$. Hence we can also apply (B-SUBMIT) in B_2 and all properties except for property 3 and 8 follow immediately. Let l_D be the integrity label of the DOM We distinguish two cases:
  - $I(\ell_a) \not\sqsubseteq_I l_D$ Then property 3 follows immediately. For property 8 the only non-trivial conditions are the claim on the parameters and the cookies of the produced event. The claim on the parameters follows directly from $=_{\bot_I}$ on the DOM and $\vec{a} = \vec{a'}$ and the claim of the cookies follows immediately from $K =_{\Gamma,\bot_I} K'$.
  - $I(\ell_a) \sqsubseteq_I l_\alpha$ the claim is trivial.

□

Next, we show the same property for browsers sending out a request of high sync integrity.

Lemma 38 (High Sync Integrity Browser Request)**.** *Let* $B_1 = (N, M, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ *and* $B_2 = (N', M', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$ *be browsers with* $B_1 \approx^B_\Gamma B_2$ *and let* $B_1 \xrightarrow{\alpha} B_1'$ *with* $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$ *and* $\alpha = \overline{\mathsf{req}}(\iota_b, n, u, p, ck, o)^{l_\alpha,\mu_\alpha}$ *Then there exist* B_2' *and* α' *such that* $B_2 \xrightarrow{\alpha'} B_2'$ *and* $\alpha =_{\bot_I} \alpha'$ *and* $B_1' \approx^B_\Gamma B_2'$.

Proof. We know that rule (B-FLUSH) was used and we know that $Q = \{\alpha\}$.

We then know by $B_1 \approx^B_\Gamma B_2$ that if $Q' = \{\alpha'\}$ with $\alpha =_{\bot_I} \alpha'$.

We can thus also apply rule (B-FLUSH) in B_2 and all claims follows immediately. □

Next we show the same property for high integrity browsers receiving a response of high sync integrity.

Lemma 39 (High Sync Integrity Browser Response)**.** *Let* B_1, B_2 *be browsers with* $B_1 \approx^B_\Gamma B_2$ *and let* $B_1 \xrightarrow{\alpha} B_1'$ *with* $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$ *and* $\alpha = \mathsf{res}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l_\alpha,\mu_\alpha}$ *with* $I(\ell_a) \not\sqsubseteq_I l_\alpha$. *Let* $\alpha' = \mathsf{res}(\iota_b, n, u, u', \vec{v'}, ck', page', s')^{l_\alpha,\mu_\alpha}$ *with* $\alpha' =_{\bot_I} \alpha$, $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$ *and* $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha'$. *Then there exist* B_2' *and such that* $B_2 \xrightarrow{\alpha'} B_2'$ *and* $B_1 \approx^B_\Gamma B_2'$

Proof. We show that all properties of Definition 16 are fulfilled.

In all cases property 6 follows immediately from Lemma 8.

Let $B_1 = (N, M, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$ and $B_2 = (N', M', P', T', Q', \vec{a'})^{\mathsf{usr},l',\mu'}$

Because of $I(\ell_a) \not\sqsubseteq_I l_\alpha$ we then know $I(\ell_a) \not\sqsubseteq_I l$,since the integrity label can not be raised between the request and the response, and $I(\ell_a) \not\sqsubseteq_I l'$ by inspection of the possible rules.

We perform a case distinction between the three possible rules.

- (B-RECVLOAD) Then because of $N = N'$, $dom(T) = dom(T')$, we can also apply (B-RECVLOAD) for B_2. We get $B_1' \approx^B_\Gamma B_2'$ from $\alpha =_{\bot_I} \alpha'$.
- (B-RECVINCLUDE) Let $T = \{tab \mapsto s\}$ and $T' = \{tab \mapsto s'\}$ Then because of $N = N'$, $s =_{\Gamma,\bot_I} s'$, we can also apply (B-RECVINCLUDE) for B_2. We get $B_1' \approx^B_\Gamma B_2'$ from $\alpha =_{\bot_I} \alpha'$.
- (B-REDIRECT) Then because of $N = N'$, , we can also apply (B-REDIRECT) for B_2. We get $B_1' \approx^B_\Gamma B_2'$ from $\alpha =_{\bot_I} \alpha'$ and $K =_{\Gamma,\bot_I} K$.

□

The next lemma treats the case, where a browser receives a response that is of high sync integrity, but of low integrity,

Lemma 40 (High Sync Integrity Browser Response of Low Integrity)**.** *Let B_1, B_2 be browsers with $B_1 \approx_\Gamma^B B_2$ and let $B_1 \xrightarrow{\alpha} B_1'$ with $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$ and $\alpha = \mathsf{res}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l_\alpha, \mu_\alpha}$ with $I(\ell_a) \sqsubseteq_I l_\alpha$. Let $\alpha' = \mathsf{res}(\iota_b, n, u, u', \vec{v'}, ck', page', s')^{l_\alpha, \mu_\alpha}$ with $\alpha =_{\perp_I} \alpha'$, $\Gamma \vDash_{\ell_a, \mathsf{usr}} \alpha$ and $\Gamma \vDash_{\ell_a, \mathsf{usr}} \alpha'$. Then there exist B_2' and such that $B_2 \xrightarrow{\alpha'} B_2'$ and $B_1 \approx_\Gamma^B B_2'$*

Proof. We show that all properties of Definition 16 are fulfilled.

In all cases property 6 follows immediately from Lemma 8.

Let $B_1 = (N, M, P, T, Q, \vec{a})^{\mathsf{usr}, l, \mu}$ and $B_2 = (N', M', P', T', Q', \vec{a'})^{\mathsf{usr}, l', \mu'}$.

We perform a case distinction between the three possible rules.

- (B-RecvLoad) Then we know that $I(\ell_a) \not\sqsubseteq_I l$ Then because of $B_1 \approx_\Gamma^B B_2$ we get that $N = N'$, $dom(T) = dom(T')$ and we can also apply (B-RecvLoad) or (B-Redirect) for B_2. Because of $I(\ell_a) \sqsubseteq_I l_alpha$ we immediately get $B_1' \approx_\Gamma^B B_2'$.
- (B-RecvInclude) Then we know that $I(\ell_a) \not\sqsubseteq_I l$. As a high integrity script cannot receive a low integrity response, this case is impossible
- (B-Redirect) Then we know that $I(\ell_a) \not\sqsubseteq_I l$ Then because of $B_1 \approx_\Gamma^B B_2$ we get that $N = N'$, $dom(T) = dom(T')$ and we can also apply (B-RecvLoad) or (B-Redirect) for B_2. Because of $I(\ell_a) \sqsubseteq_I l_alpha$ we immediately get $B_1' \approx_\Gamma^B B_2'$. Then because of $N = N'$, we can also apply (B-Redirect) or (B-Load) for B_2.

□

Next we show the same property for servers taking an internal step of high sync integrity.

Lemma 41 (High Sync Integrity Server Steps)**.** *Let S_1, S_2 be servers with $S_1 \approx_\Gamma^S S_2$ and let S_2 be deterministically terminating. Let $S_1 \xrightarrow{\alpha} S_1'$ with $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$ and $\alpha \in \{\bullet, \sharp[\cdot]^\cdot_\cdot\}$. Then $bad(S_1')$ or there exist S_2' and α', β such that $S_2 \xrightarrow{\beta \cdot \alpha'} {}^* S_2'$ with $I(\ell_a) \sqsubseteq_I sync_I(\beta_k)$ for all $\beta_k \in \beta$ and $\alpha =_{\perp_I} \alpha'$ and $S_1' \approx_\Gamma^S S_2'$.*

Proof. Let $S_1 = (D_1, \phi_1, t_1^0)$, $S_1' = (D_1', \phi_1', t_1^{0'})$, $S_2 = (D_2, \phi_2, t_2^0)$, $S_2' = (D_2', \phi_2', t_2^{0'})$. Then there is $t_1 \in \text{running}(S_1)$ with $(D_1, \phi_1, t_1) \xrightarrow{\alpha} (D_1', \phi_1', t_1')$ and $t_1' \in \text{running}(S_1')$. Let $t_1 = \lceil c_1 \rfloor_{E_1, R_1}^{l_1, \mu_1}$ and $t_1' = \lceil c_1' \rfloor_{E_1', R_1}^{l_1', \mu_1}$.

Because of $I(\ell_a) \not\sqsubseteq_I \alpha$ we know that $I(\ell_a) \not\sqsubseteq_I int_\sqcap(t)$ and by the definition of $\approx_\Gamma^S$ we know that there exists a corresponding thread $c(t_1) = t_2 \in \text{running}(S_2)$ with $t_2 = \lceil c_2 \rfloor_{E_2, R_2}^{l_2, \mu_2}$.

We now show that there are α, β and $t_2' = \lceil c_2' \rfloor_{E_2', R_2}^{l_2', \mu_2}$, $t_2'' = \lceil c_2'' \rfloor_{E_2'', R_2}^{l_2'', \mu_2}$ with $(D_2, \phi_2, t_2) \xrightarrow{\beta} {}^*(D_2'', \phi_2'', t_2'') \xrightarrow{\alpha} (D_2', \phi_2', t_2')$.

Let $S_2'' = (D_2'', \phi_2'', t_2^{0''})$.

We perform the proof by induction over the derivation of the step α.

For all cases except (S-RESET) we let $\beta = \epsilon$ and $S_2'' = S_2$.

- (S-SEQ) Claim follows by induction.
- (S-IFTRUE) Then $c_1 = \textbf{if } se \textbf{ then } c_{11} \textbf{ else } c_{12}$ and $c_2 = \textbf{if } se' \textbf{ then } c_{21} \textbf{ else } c_{22}$ with $se =_{\perp_I} se'$. Let $v^\tau = eval_{E_1}(se, D_1)$ and $v'^{\tau'} = eval_{E_2}(se', D_2)$. Then by Lemma 27 we get $v^\tau =_{\perp_I} v'^{\tau'}$. We distinguish two cases:
  - If reply, redir, tokencheck, origincheck $\in coms(c_{11}) \cup coms(c_{12})$. We distinguish to cases
    - If $I(\ell_a) \not\sqsubseteq_I I(\tau)$ then we also have $I(\ell_a) \not\sqsubseteq_I I(\tau')$ and we have $v = v'$. Hence the continuations are $c_1 = c_{11}$ and $c_2 = c_{21}$ and the claim follows immediately.
    - If $I(\ell_a) \sqsubseteq_I I(\tau)$ then we also have $I(\ell_a) \sqsubseteq_I I(\tau')$. We hence have $I(\ell_a) \sqsubseteq_I l_1'$ and $I(\ell_a) \sqsubseteq_I l_2'$ and the claim follows.
  - If reply, redir, tokencheck, origincheck $\notin coms(c_{11}) \cup coms(c_{12})$. We distinguish to cases
    - If $I(\ell_a) \not\sqsubseteq_I I(\tau)$ then we also have $I(\ell_a) \not\sqsubseteq_I I(\tau')$ and we have $v = v'$. Hence the continuations are c_{11}; reset l and c_{21}; reset l and the claim follows immediately.
    - If $I(\ell_a) \sqsubseteq_I I(\tau)$ then we also have $I(\ell_a) \sqsubseteq_I \tau'$. Then $t_1' = \lceil c_{12}; \text{reset } l_1 \rfloor_{E_1,R_1}^{l_1 \sqcup_I I(\tau), \mu_1}$ and $t_2' = \lceil c_2'; \text{reset } l_2 \rfloor_{E_2,R_2}^{l_2 \sqcup_I I(\tau'), \mu_2}$ where $c_2' \in \{c_{11}', c_{12}\}$. The claim then follows immediately.
- (S-FALSE) This case is analog to the case of rule (T-TRUE).
- (S-TOKENCHECKTRUE), Then $c_1 = \textbf{if tokenchk}(se_{11}, se_{12}) \textbf{ then } c_1''$ and $c_2 = \textbf{if tokenchk}(se_{21}, se_{22}) \textbf{ then } c_2''$.

  Let $v_{11} = eval_{E_1}(se_{11}, D_1), v_{12} = eval_{E_1}(se_{12}, D_1), v_{21} = eval_{E_2}(se_{21}, D_2), v_{22} = eval_{E_2}(se_{22}, D_2)$

  We then know that $v_{11} = v_{12}$.

  We distinguish two cases:
  - if $v_{21} = v_{22}$ then $c_1 = c_1''$ and $c_2 = c_2''$ and the claim is trivial.
  - if $v_{21} \neq v_{22}$ then we have $c_2' = \textbf{reply}\ (\text{error}, \textbf{skip}, \{\})$. This however is a contradiction to the assumption of the deterministic termination
- (S-TOKENCHECKFALSE) Then $t_1' = \lceil \textbf{reply}\ (\text{error}, \textbf{skip}, \{\}) \rfloor_{E_1,R_1}^{l_1,\mu_1}$ and the claim is trivial, since we have $bad(S_1')$
- (S-SKIP) Trivial

- (S-RESET) We distinguish two cases:
  - If $I(\ell_a) \not\sqsubseteq_I l_1$ then we have $l_1 = l_2$ and the claim is trivial.
  - Otherwise we know that $c_1 = \text{reset } l_r$ where $I(\ell_a) \not\sqsubseteq_I l_r$. Then we know by property 2c of Definition 18 that $c_2 = c_{2r}; \text{reset } l_1; c'_{2r}$ for some c_{2r}, c'_{2r}, with $c'_1 =_{\perp_I} c'_{2r}$. We then let $c'' = \text{reset } l_1; t'_{2r}$ and $c' = t'_{2r}$ and show that they fulfill the claim.
    Since we know that $\mathbf{reply}, \mathbf{redir}, \mathbf{tokencheck}, \mathbf{origincheck} \notin coms(c_2)_r$, we know by deterministic termination of t_2 that $t_2 \xrightarrow{\beta}{}^* \lceil c'' \rfloor^{l'',\mu}_{R'',E''}$.
    By repeated application of Lemma 30 and Lemma 26 we get $S_1 \approx^S_\Gamma S''_2$.
    Now we need to show $S'_1 \approx^S_\Gamma S'_2$. All claims from Definition 18 except for property 2c are trivial. For property 2c $l'_1 = l'_2$ follows immediately from rule (S-RESET) and $c'_1 =_{\perp_I} c'_2$ follows immediately from $c'_1 =_{\perp_I} c'_{2r}$.
- (S-RESTORESESSION) Then $c_1 = \mathbf{start}\ se$ and $c_2 = \mathbf{start}\ se'$, with $se =_{\perp_I} se'$. Then for S_2 we can apply rule (S-RESTORESESSION) or (S-NEWSESSION) and the claim follows because using Lemma 27 we immediately get $j'_1 =_{\perp_I} j'_2$.
- (S-NEWSESSION) Analog to previous case.
- (S-SETGLOBAL) We have $c_1 = r := se$ and $c_2 = r := se'$ with $se =_{\perp_I} se'$. The claim then follows immediately using Lemma 27.
- (S-SETSESSION) This case follows analog to the previous one.
- (S-LOGIN) We have $c_1 = \mathbf{login}\ se_1, se_2, se_3$ and $c_2 = \mathbf{login}\ se'_1, se'_2, se'_3$ with $se_1 =_{\perp_I} se'_1$, $se_2 =_{\perp_I} se'_2$ and $se_3 =_{\perp_I} se'_3$. Let $j^\tau_1 = eval_{E_1}(se_3, D_1)$ and let $j^{\tau'}_2 = eval_{E'_2}(se'_3, D_2)$. We distinguish two cases:
  - If $I(\ell_a) \sqsubseteq_I I(\tau)$ then also $I(\ell_a) \sqsubseteq_I I(\tau')$ and the claim follows immediately.
  - If $I(\ell_a) \not\sqsubseteq_I I(\tau)$ then $j^\tau_1 = j'^{\tau'}_1$. By rule (T-LOGIN) and Lemma 5 we know that $\tau = \texttt{cred}(\ell)$. Let $v^{\tau_1}_1 = eval_{E_1}(se_1, D_1)$, let $v'^{\tau'_1}_1 = eval_{E_1}(se'_1, D_1)$, let $v^{\tau_2}_2 = eval_{E_1}(se_2, D_1)$ and let $v'^{\tau'_2}_2 = eval_{E_2}(se'_2, D_2)$. Then by rule (T-LOGIN) and Lemma 5 we know that and $I(\ell_a) \not\sqsubseteq_I \tau_1$ and $I(\ell_a) \not\sqsubseteq_I \tau_2$ and hence by $se_1 =_{\perp_I} se'_1$ and $se_2 =_{\perp_I} se'_2$ we get $v^{\tau_1}_1 = v'^{\tau'_1}_1$ and $v^{\tau_2}_2 = v'^{\tau'_2}_2$. With $\iota_b = eval_{E_1}(se_1, D_1)$ and let $\iota_b{}' = eval_{E_2}(se'_1, D_2)$ we get using the properties of ρ
- (T-AUTH) Then we have $c_1 = \mathbf{auth}\ \vec{se_1}\ \mathbf{at}\ \ell$ and $c_2 = \mathbf{auth}\ \vec{se_2}\ \mathbf{at}\ \ell$ with $\vec{se_1} =_{\perp_I} \vec{se_2}$. If $I(\ell_a) \sqsubseteq_I \ell$, then the claim is trivial. We hence assume $I(\ell_a) \not\sqsubseteq_I \ell$.
  We then know by rule (T-AUTH) that $I(\ell_a) \not\sqsubseteq_I l$.
  Let $v^{\tau_{1,i}}_{1,i} = eval_{E_1}(se_{1,i}, D_1)$ and Let $v^{\tau_{2,i}}_{2,i} = eval_{E_2}(se_{2,i}, D_2)$. Let $R_1 = R_2 = n, u, \iota_b, o$, let $E_1 = i_1, j_1$ and $E_2 = i_2, j_2$, and let $\iota_{s1} = \phi(j_1)$ and $\iota_{s2} = \phi(j_2)$.
  We then know $\vec{se_1} =_{\perp_I} \vec{se_2}$ and $j_1 =_{\perp_I} j_2$
  We have $\alpha = \sharp[\vec{v_1}]^{\iota_b,\iota_{s1}}_\ell$ and $\alpha' = \sharp[\vec{v_2}]^{\iota_b,\iota_{s2}}_\ell$.

By rule (T-AUTH) we know that $I(\ell_a) \not\sqsubseteq_I I(\tau_{1,i})$ and $I(\ell_a) \not\sqsubseteq_I I(\tau_{2,i})$, we thus have $v_1^{\vec{\tau}_1} = v_2^{\vec{\tau}_2}$ by Lemma 27.

By rule (T-AUTH) we also know that that $I(\ell_a) \not\sqsubseteq_I I(jlabel(j_1))$ and $I(\ell_a) \not\sqsubseteq_I I(jlabel(j_2))$. We thus get by property 4 of Definition 18 that $\iota_{s1} = \iota_{s2}$.

We thus have $\alpha = \alpha$ and the claim follows.

- (S-OCHECKSUCC) We then have $c_1 = \textbf{if originchk}(O) \textbf{ then } c_1'$ and $c_2 = \textbf{if originchk}(O) \textbf{ then } c_2'$. With $R_1 = R_2 = n, u, \iota_b, o$ we know that we can also apply rule (S-OCHECKSUCC) in t_2 and the claim follows immediately.
- (S-OCHECKFAIL) We then have $c_1 = \textbf{if originchk}(O) \textbf{ then } c_1'$ and $c_2 = \textbf{if originchk}(O) \textbf{ then } c_2'$. With $R_1 = R_2 = n, u, \iota_b, o$ we know that we can also apply rule (S-OCHECKFAIL) in t_2 which contradicts our assumption about the termination of t_2. This case is thus impossible

□

Next, we show the same property for servers receiving a request of high sync integrity.

Lemma 42 (High Sync Integrity Server Request)**.** *Let S_1, S_2 be servers with $S_1 \approx_\Gamma^S S_2$ and let S_2 be the corresponding server of S_1 as defined in Definition 19. Let $S_1 \xrightarrow{\alpha} S_1'$ with $I(\ell_a) \not\sqsubseteq_I \alpha$ and $\alpha = \mathsf{req}(\iota_b, n, u, p, ck, o)^{l,\mu}$. Let α' with $\alpha =_{\perp_I} \alpha'$ and $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha$, $\Gamma \vDash_{\ell_a,\mathsf{usr}} \alpha'$. Then there exist S_2' such that $S_2 \xrightarrow{\alpha'} S_2'$ and $S_1' \approx_\Gamma^S S_2'$.*

Proof. Then the step is taken using rule (S-RECV), We thus have $t = u[\vec{r}](\vec{x}) \hookrightarrow c \in threads(S_1)$. Because S_2 is the corresponding server of S_1 we know that $t \in threads(S_2)$.

We can thus apply rule (S-RECV) in and take the step $S_2 \xrightarrow{\alpha'} S_2'$.

We now show $S_1' \approx_\Gamma^S S_2'$ by showing the properties of Definition 18.

- Property 1 follows immediately from Lemma 17
- Property 2a is trivial (For simplicity we assume that sampling returns the same result on both servers)
- Property 2b follows immediately from the claim on cookies in $\alpha =_{\perp_I} \alpha$.
- Property 2c follows from the claim on the parameters in $\alpha =_{\perp_I} \alpha$.
- Properties 3 and 4 are trivial since the session memory and trust mapping are not modified in rule (S-RECV).

Let t and t' be the freshly generated running threads. Then $t =_{\perp_I} t'$ follows from the claim on p in $\alpha =_{\perp_I} \alpha'$. □

Next, we show the same property for servers sending a response of high sync integrity.

Lemma 43 (High Sync Integrity Server Response)**.** *Let S_1, S_2 be servers with $S_1 \approx^S_\Gamma S_2$ and let $S_1 \xrightarrow{\alpha} S_1'$ with $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$ and $\alpha = \overline{\mathsf{res}}(\iota_b, n, u, u', \vec{v}, ck, page, s)^{l,\mu}$ Then there exist S_2' and α' such that $S_2 \xrightarrow{\alpha'} S_2'$ and $S_1' \approx^S_\Gamma S_2'$ and $\alpha =_{\bot_I} \alpha'$.*

Proof. We distinguish two cases for the rule applied to take the step:

- (S-REPLY) We have $c = \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se}$ and $c' = \mathbf{reply}\ (page, s, ck)\ \mathbf{with}\ \vec{x} = \vec{se'}$ with $\forall i \in [1 \dots |\vec{se}|] se_i =_{\bot_I} se_i'$.

  Let $v_i = eval_E(se_i, D)$ and $v_i' = eval_{E'}(se_i', D')$. Then by Lemma 27 we get $v_i =_{\bot_I} v_i'$.

  With $\sigma = \{x_1 \mapsto v_1 \cdots x_m \mapsto v_m\}$ and $\sigma' = \{x_1 \mapsto v_1' \cdots x_m \mapsto v_m'\}$

  We immediately get $s\sigma =_{\bot_I} s\sigma'$, $page\sigma =_{\bot_I} page\sigma'$ and $ck\sigma =_{\Gamma,\bot_I} ck'$.

  The claim then follows using Lemma 18.

- (S-REDIR) We have $c = \mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se}$ and $c' = \mathbf{redirect}\ (u, \vec{z}, ck)\ \mathbf{with}\ \vec{x} = \vec{se'}$ with $\forall i \in [1 \dots |\vec{se}|] se_i =_{\bot_I} se_i'$.

  Let $v_i = eval_E(se_i, D)$ and $v_i' = eval_{E'}(se_i', D')$. Then by Lemma 27 we get $v_i =_{\bot_I} v_i'$.

  With $\sigma = \{x_1 \mapsto v_1 \cdots x_m \mapsto v_m\}$ and $\sigma' = \{x_1 \mapsto v_1' \cdots x_m \mapsto v_m'\}$

  We immediately get $\vec{z}\sigma =_{\bot_I} \vec{z'}\sigma'$ and $ck\sigma =_{\bot_I} ck'\sigma$.

  The claim then follows using Lemma 18.

□

Finally, we show the same property on websystem level.

Lemma 44 (High Sync Integrity Steps)**.** *Let $A_1 = (\mathcal{K}_1, \ell_a) \triangleright W_1$ and $A_2(\mathcal{K}_2, \ell_a) \triangleright W_2$ be web systems with $A_1 \approx_\Gamma A_2$ and let A_2 be deterministically terminating. Then whenever $A_1 \xrightarrow{\alpha} A_1'$ with $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$ then $bad(A_1')$ or there exist $\vec{\beta}$, α' and A_2' such that $A_2 \xrightarrow{\vec{\beta}\cdot\alpha'} A_2'$ with $\alpha =_{\bot_I} \alpha'$ and for all $\beta \in \vec{\beta}$ we have $I(\ell_a) \sqsubseteq_I sync_I(\beta)$ and $A_1' \approx_\Gamma A_2'$.*

Proof. If A_2 is not in a low integrity state as defined in Lemma 36, then let $A_2'' = A_2$. Otherwise, let $A_2'' = \mathbf{nexth}(A_2)$ as in Lemma 36. We then know that $A_2 \xrightarrow{\vec{\beta}} A_2''$ where for $\beta \in \vec{\beta}$ we have $I(\ell_a) \sqsubseteq_I \text{sync}_I(\beta)$ and $A_2 \approx_\Gamma A_2''$. By transitivity we hence get $A_1 \approx_\Gamma A_2''$. We furthermore know that A_2'' is in one of the five states described in Lemma 36.

We now show that $A_2'' \xrightarrow{\alpha'} A_2'$ with $\alpha =_{\bot_I} \alpha'$ and $A_2 \approx_\Gamma A_2'$. We prove the claim by induction over the derivation of the step α.

- (A-NIL) Then, if the step is derived through rule (W-LPARALLEL) or (W-RPARALLEL) the claim follows by induction. The claim for internal server steps follows from Lemma 41. For internal browser steps, we perform a case distinction: Let browsers$(W) \ni B = (N, K, P, T, Q, \vec{a})^{\mathsf{usr},l,\mu}$
  - if $I(\ell_a) \not\sqsubseteq_I l$ then the claim follows immediately from Lemma 37.
  - if $I(\ell_a) \sqsubseteq_I l$ then we know that rule (B-END) is used. We know that $A_2'' = \mathbf{nexth}(A_2)$ is in one of the five states described in Lemma 36. Since we already excluded one possible state, and three other states require $I(\ell_a) \not\sqsubseteq_I l$, we know that the browser B_2'' in A_2'' is in a state where rule (B-END) can be used. The claim then follows immediately.
- (A-BROWSERSERVER) Then we can also apply (A-BROWSERSERVER) for A_2'' and the claim follows from Lemma 38 for the browser step and Lemma 42 for the server in case of a high integrity request or Lemma 33 for the server in case of a low integrity request.
- (A-SERVERBROWSER) Then we distinguish two cases
  - Integrity of the response is high: Then we can also apply rule (A-SERVERBROWSER) in $A_2'' = A_2$ and the claim follows from Lemma 43 for the server and Lemma 39 for the browser step.
  - Integrity of the response is low: Then we know by Lemma 36 that $A_2'' = \mathbf{nexth}(A_2)$ is in a state where the browser can receive a request and the server can send a reply or a redirect or a timeout response is ready to be sent. We immediately get $\alpha =_{\perp_I} \alpha'$. Then we can also apply rule (A-SERVERBROWSER) and the claim follows from Lemma 30 for the server step and from Lemma 40 for the browser step.
- (A-TIMEOUTSEND) Then we can also apply (A-TIMEOUTSEND) in A_2'' and the claim follows from Lemma 38 for the browser step in case of a high integrity browser state or from Lemma 31 in case of a low integrity browser state.
- (A-TIMEOUTRECV) Then we can also apply (A-TIMEOUTRECV) in A_2" and the claim follows from Lemma 39 or Lemma 31 for the browser step.
- (A-BROATK) Then we distinguish two cases
  - W_2 can perform a step using (A-BROWSERSERVER): Then the claim follows from Lemma 38 or Lemma 31 for the browser step and Lemma 30 for the server.
  - W_2 can perform a step using (A-TIMEOUTSEND) Then the claim follows from Lemma 38 or Lemma 31 for the browser step.
- (A-ATKSER) Cannot happen, event is of high integrity
- (A-SERATK) Cannot happen, event is of high integrity
- (A-ATKBRO) Then we distinguish two cases:

- W_2 can perform a step using (A-SERVERBROWSER): Then the claim follows from Lemma 39 for the browser step.
- W_2 can perform a step using (A-TIMEOUTRECV) Then the claim follows from Lemma 39 for the browser step.

□

Using the previous lemmas, we can conclude that the relation $\approx_\Gamma$ fulfills core properties, that will allow us to prove the main theorem.

Lemma 45. *Let A_1 and A_2 be web systems with $A_1 \approx_\Gamma A_2$. Then*

1. *$bad(A_1)$*
2. *or the following properties hold:*
   a) *if $A_1 \overset{\alpha}{\Rightarrow}_\Gamma A_1'$ and $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$ then there exists $\alpha', \vec{\beta}$ and A_2' such that*
      - *for all $\beta \in \vec{\beta}$ we have $I(\ell_a) \sqsubseteq_I sync_I(\beta)$*
      - *$A_2 \overset{\vec{\beta}\cdot\alpha'}{\Longrightarrow}_\Gamma {}^* A_2'$*
      - *$\alpha =_{\bot_I} \alpha'$*
      - *$A_1' \approx_\Gamma A_2'$*
   b) *if $A_1 \overset{\alpha}{\Rightarrow}_\Gamma A_1'$ for some α with $I(\ell_a) \sqsubseteq_I sync_I(\alpha)$ then $A_1' \approx_\Gamma A_2$.*

Proof. If $bad(A_1)$ then the claim is trivial. We hence assume $\neg bad(A_1)$, which then immediately gives us $A_1 \approx_\Gamma A_2$. The claim for the low integrity step then follows immediately from Lemma 35 and the transitivity of $\approx_\Gamma$ (Lemma 28) and the claim for the high integrity step follows from Lemma 44. □

Intuitively,the relation fulfills the following properties: Either the first websystem is in a bad state, or

1. Whenever the first system takes a step of high sync integrity, then the second system can take a number of steps of low sync integrity, followed by the same step of high sync integrity, and the resulting websystems are in the relation.
2. If the first system takes a step of low sync integrity, then it remains in relation with the second system (which didn't take a step).

B.2.6 Main Theorem

In this section we bring together the results from the previous sections in order to show our main theorem.

First, we show that whenever an attacked and an unattacked websystem are in the relation $\approxeq_\Gamma$, and the attacked system generates a trace, then the unattacked websystem can generate a trace that has the same events with high sync integrity,

Lemma 46 (High Integrity Trace Equality)**.** *Let $high(\gamma)$ be the trace containing only the events $\alpha \neq \bullet$ with $I(\ell_a) \not\sqsubseteq_I sync_I(\alpha)$.*

Let A_1 be an attacked and A_2 and unattacked websystem with $A_1 \approxeq_\Gamma A_2$. Then if A_1 generates the trace γ_1, then A_2 can generate a trace γ_2 such that $high(\gamma_1) = high(\gamma_2)$.

Proof. We prove the claim by induction over the generated trace γ, using the properties of $\approxeq_\Gamma$ from Lemma 45

1. If $\gamma_1 = \epsilon$ then the claim is trivially fulfilled.
2. If $\gamma_1 = \alpha \cdot \gamma_1'$: then A_1 takes the step α to reach state A_1' and produces the trace $\alpha \cdot \gamma_1'$. If $bad(A_1)$ then we know that $high(\alpha \cdot \gamma) = \epsilon$, since according to Definition 15 we either have $\alpha = \bullet$ or $I(\ell_a) \sqsubseteq_I int(\alpha)$ and the claim is trivial. We hence assume $\neg bad(A_1)$ and hence know $A_1 \approx_\Gamma A_2$.

   We distinguish two cases

   a) If $I(\ell_a) \not\sqsubseteq_I \text{sync}_I(\alpha)$ then by Lemma 45 A_2 can take the steps $\vec{\beta} \cdot \alpha$, where $I(\ell_a) \sqsubseteq_I \text{sync}_I(\beta)$ for all $\beta \in \vec{\beta}$ and hence produces the trace $\vec{\beta} \cdot \alpha \cdot \gamma_2'$. We hence have $high(\vec{\beta} \cdot \alpha) = \alpha$. Since $A_1' \approx_\Gamma A_2'$, we can apply the induction hypothesis and get $high(\gamma_1') = high(\gamma_2')$, hence we also have $high(\gamma_1) = high(\alpha \cdot \gamma_1') = high(\alpha \cdot \gamma_2') = high(\gamma_2)$.

   b) If $I(\ell_a) \sqsubseteq_I \text{sync}_I(\alpha)$ then we have $high(\gamma_1') = high(\gamma_1)$ and since by Lemma 45 we know $A_1' \approxeq_\Gamma A_2$, we can apply the induction hypothesis and get $high(\gamma_2) = high(\gamma_1') = high(\gamma_1)$.

□

Next, we show that whenever a well-typed websystem produces a high integrity authenticated event then it also has high sync integrity.

Lemma 47 (High Integrity Auth Events)**.** *Let* usr *be the honest user and for all u with $\rho(\mathsf{usr}, u) = n^\tau$ we have $C(\tau) \not\sqsubseteq_C C(\ell_a)$. Let ℓ_a be an attacker. For any A with $\Gamma \vDash_{\ell_a,\mathsf{usr}} A$ and $A \xrightarrow{\vec{\alpha}}{}^* A'$, if for $\beta = \sharp[\vec{v}]_\ell^{\iota_b,\iota_u}$ we have $\beta \in \vec{\alpha}$ and $I(\ell_a) \not\sqsubseteq_I I(\ell)$ and $\iota_b =$ usr or $\iota_u =$ usr then we have $I(\ell_a) \not\sqsubseteq_I sync_I(\beta)$,*

Proof. Let $l = \text{sync}_I(\beta)$.

There exist A_1, A_2 such that $A \xrightarrow{\vec{\alpha_1}}{}^* A_1 \xrightarrow{\beta} A_2 \xrightarrow{\vec{\alpha_2}}{}^* A'$ with $\Gamma \vDash_{\ell_a,\mathsf{usr}} A_1$ by Lemma 23.

We also know that $S_1 \in \text{servers}(A_1), S_2 \in \text{servers}(A_2)$ with $S_1 \xrightarrow{\beta} S_2$, with $\Gamma \vDash_{\ell_a,\mathsf{usr}} S_1$ by Definition 13

Let $S_1 = (D, \phi, t)$. Then there is $\lceil c \rfloor^{l,\mu}_{R,E} \in \text{running}(S_1)$ with $(D, \phi, \lceil c \rfloor^{l,\mu}_{R,E}) \xrightarrow{\beta} (D, \phi, \lceil c' \rfloor^{l,\mu}_{R,E})$ for some c, c'.

We know that the event β is produces using rule (S-AUTH), hence have

$$(D, \phi, \lceil \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell \rfloor^{l,\mu}_{R,E}) \xrightarrow{\beta} (D, \phi, \lceil \mathbf{skip} \rfloor^{l,\mu}_{R,E})$$

with $R = n, u, \iota_b, o$ and $E = i, j$ with $\phi(j) = \iota_u$.

Let

$$b = \begin{cases} \mathsf{att} & \text{if } \iota_b \neq \mathsf{usr} \\ \mathsf{hon} & \text{if } \mu = \mathsf{hon} \wedge \iota_b = \mathsf{usr} \\ \mathsf{csrf} & \text{if } \mu = \mathsf{att} \wedge \iota_b = \mathsf{usr} \end{cases}$$

$$\Gamma'_{\mathcal{R}^@} = \begin{cases} \Gamma_{\mathcal{R}^@} & \text{if } (\iota_b = \mathsf{usr}) \\ \{\_ \mapsto \ell_a\} & \text{if } (\iota_b \neq \mathsf{usr}) \end{cases}$$

$$\Gamma' = (\Gamma_{\mathcal{U}}, \Gamma_{\mathcal{X}}, \Gamma'_{\mathcal{R}^@}, \Gamma_{\mathcal{R}^\$}, \Gamma_{\mathcal{V}})$$

We then get by rule (T-RUNNING)

$$\Gamma', jlabel(j), l \vdash^{\mathsf{c}}_{\ell_a,(u,b,\mathcal{P})} \mathbf{auth}\ \vec{se}\ \mathbf{at}\ \ell : \_, l$$

We distinguish two cases:

- If $b \neq \mathsf{att}$ then by typing we know from rule (T-AUTH) that we have $I(\ell_a) \not\sqsubseteq_I l$ and the claim follows immediately.
- If $b = \mathsf{att}$ then we know that $\iota_b \neq \mathsf{usr}$. Hence we must have $\phi(j) = \iota_u = \mathsf{usr}$. We now show that this case can also not happen. Because of $C(\rho(\iota_u)) \not\sqsubseteq_C C(\ell_a)$ and property 3 of Definition 12 we know that $C(\rho(\iota_u)) \sqsubseteq_C C(jlabel(j))$. Since an attacker can never have a session with a high confidentiality session label, we know $C(jlabel(j)) \sqsubseteq_C C(\ell_a)$ and we immediately have a contradiction.

□

We define a well-formed attacker to be an attacker whose knowledge is limited by his label.

Definition 20 (Well-formed attacker)**.** *An attacker* $(\ell_a, \mathcal{K})$ *is* well-formed *if* $\forall n^\tau \in \mathcal{K}$ *we have* $\tau \sqsubseteq_{\ell_a} \ell_a$.

This lemma shows that the initial state is in the relation $\approxeq_\Gamma$ with itself.

Lemma 48 (The initial state is in $\approxeq_\Gamma$)**.** *Assume a well-formed server cluster* W_0*, an honest browser of the user* usr $B_{\mathsf{usr}}(\{\}, \vec{a})$ *with well formed user actions* $\vec{a}$*, a well-formed attacker* $(\ell_a,\mathcal{K})$ *and let* $A = (\mathcal{K}, \ell_a) \triangleright B_{\mathsf{usr}}(\{\}, \vec{a}) \parallel W_0$*. If for all servers* $S = (D, \phi, t)$ *of* W_0*, we have* $\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$*, then* $\overline{A} \approxeq_\Gamma \overline{A}$.

Proof. We fist show $\overline{A} \approx_\Gamma \overline{A}$ by showing the different properties of Definition 19.

- For property 1, $\Gamma \vDash_{\ell_a,\mathsf{usr}} A$ we show that the properties of Definition 13 are fulfilled:
  - We get property 1, $\Gamma \vDash_{\ell_a,\mathsf{usr}} \overline{B_{\mathsf{usr}}(\{\}, \vec{a})}$ by checking that all the properties of Definition 11 hold. Property 6 follows from the well-formedness of $\vec{a}$. All other properties are trivial,
  - We get 2, $\Gamma \vDash_{\ell_a,\mathsf{usr}} \overline{S}$ for all servers $S = (D, \phi, t) \in \text{servers}(W_0)$ by checking that all properties of Definition 12. Property 4 follows from $\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$, using Lemma 2. All other properties are trivial for fresh servers (as defined in Definition 5).
  - Property 3 is trivial since there are no network connections in the browser.
  - Property 4 follows immediately from the well-formedness of the attacker.
- property 2 is trivial
- For property 3, $\overline{S} \approx^S_\Gamma \overline{S}$ we show that the properties of Definition 18 hold:
  - property 1 follows immediately from $\Gamma \vDash_{\ell_a,\mathsf{usr}} \overline{A}$, which we have already shown.
  - All other properties are trivial for fresh servers.
- For property 3, $\overline{B_{\mathsf{usr}}(\{\}, \vec{a})} \approx^B_\Gamma \overline{B_{\mathsf{usr}}(\{\}, \vec{a})}$ we show that the properties of Definition 16 hold:
  - property 1 follows immediately from $\Gamma \vDash_{\ell_a,\mathsf{usr}} \overline{A}$, which we have already shown.
  - All other properties are trivial for fresh browsers.

□

Our main theorem states that typing ensures web session integrity – if we consider all ingredients to be well-formed.

Theorem 2 (Typing implies Web Session Integrity)**.** *Let W be a fresh cluster, (ℓ_a,$\mathcal{K}$) a well-formed attacker, Γ^0 a typing environment with $\lambda, \ell_a, \Gamma^0 \vdash \diamond$ and let $\vec{a}$ be a list of well-formed user actions for* usr *in W with respect to Γ^0 and ℓ_a. Assume that for all u with $\rho(\mathsf{usr}, u) = n^\tau$ we have $C(\tau) \not\sqsubseteq_C C(\ell_a)$ and that we have $\Gamma^0 \vdash^{\mathsf{t}}_{\ell_a,\mathcal{P}} t$ for all servers $S = (\{\}, \{\}, t)$ in W. Then W preserves* session integrity *against (ℓ_a,$\mathcal{K}$) for the honest user* usr *performing the list of actions $\vec{a}$.*

Proof. Let $W' = B_{\mathsf{usr}}(\{\}, \vec{a}) \parallel W$ and let $A = (\ell_a, \mathcal{K}) \triangleright W'$.

We have to show that for any attacked trace γ generated by the attacked system A there exists a corresponding unattacked trace γ' generated by A such that

$$\forall I(\ell) \not\sqsubseteq_I I(\ell') : \gamma \downarrow (\mathsf{usr}, \ell') = \gamma' \downarrow (\mathsf{usr}, \ell')$$

By Lemma 48, we know $\overline{A} \approxeq_\Gamma \overline{A}$.

By Lemma 1 we know that also $\overline{A}$ can produce the trace α.

Applying Lemma 46 , we know that there exists an unattacked trace γ' produced by $\overline{A}$, such that $high(\gamma) = high(\gamma')$.

Since for all $\alpha = \sharp[\vec{v}]^{\iota_s}_{\ell'}$ with $\ell' \not\sqsubseteq \ell$ we have $int(\alpha) = \bot_I$ by Lemma 47, we know that

$$\forall I(\ell) \not\sqsubseteq_I I(\ell') : \gamma \downarrow (\mathsf{usr}, \ell') = \gamma' \downarrow (\mathsf{usr}, \ell')$$

By Lemma 1 we know that this trace γ' can also be produced by A.

□