TU WIEN Informatics

WebAPISpec: an Extensible, Machine Checked Model of Secure Browser Specifications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Logic and Computation

eingereicht von

Alice Lee, BSc
Matrikelnummer 12127691

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Matteo Maffei
Mitwirkung: Dott.mag. Lorenzo Veronese

Wien, 9. Mai 2024

Alice Lee	Matteo Maffei

TU WIEN Informatics

WebAPISpec: an Extensible, Machine Checked Model of Secure Browser Specifications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Logic and Computation

by

Alice Lee, BSc
Registration Number 12127691

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Matteo Maffei
Assistance: Dott.mag. Lorenzo Veronese

Vienna, May 9, 2024

Alice Lee	Matteo Maffei

Erklärung zur Verfassung der Arbeit

Alice Lee, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. Mai 2024

Alice Lee

Acknowledgements

First and foremost, I would like to thank Lorenzo Veronese for being my co-advisor for this thesis and the project that preceded it. Not only did you meet with me for weekly discussions, you also generously took the time to give me very extensive edits on two separate read-throughs of my thesis which noticeably improved not just the readability and the content but also my overall understanding of the goal of the thesis.

Thank you to my advisor Matteo Maffei for your idea to apply the DY* proof approach to web browser security and for your deep knowledge about formal methods for security, I always left my conversations with you feeling inspired about my project and the field in general. I really enjoyed working with everyone in the Security and Privacy group at TU Wien, whether we were getting Friday lunch or gathering around the espresso machine, the sense of community made me enjoy coming into the office every day.

I'd like to thank my former colleague Nabil Schear for encouraging me to see a master's degree as an opportunity to learn how to be a researcher instead of just to learn new subjects, and also for your more practical advice about organizing a research project.

I'd like to thank the friends I've made here, who I am happily surprised I found quite easily, with special thanks to Oskar for extensive edits, Felix and Philipp for allowing me to rope them into a writing group, Felicia for checking the German translation of my abstract, and my boyfriend Alex for moral support and more editing. I also want to thank my friends from home, I'm so glad that so many of you have made the time to visit me here or to make sure to see me when I'm back in Boston and make me feel like home is not so far away. Thank you Lani, for reading through the whole thing for a grammar check! Thank you Mary, for making Vienna feel like an accessible city to me.

Finally, thank you to my parents Jon and Sally and my sister Hannah for making the time to visit me here every year and supporting me spending some time on a different continent.

Abstract

Browsers are implemented based on specifications written by disjoint groups of people. Specifications for different components of the browser can end up being contradictory in a way that introduces security bugs because of the complexity of the software. One way to reduce the likelihood of such specification bugs is to build a formal model of the browser that can be mechanically checked for clashes instead of relying on people to examine informal written descriptions for potential problems.

We propose a new browser specification model that uses refinement types to check that all provided API functions maintain our defined security invariant. The model is a transition system in which the state is the browser internals and trace of events and the state transitions are the API functions. If all API functions are safe, then any script which only interacts with the browser using those functions cannot violate the invariant. The invariant is a conjunction of existing component security invariants from the literature, which we refer to as subinvariants.

This approach allows us to prove that scripts are safe without modeling the contents of the scripts themselves. We are also able to model unbound traces without the exponential blow up of more traditional state space exploration approach. In addition, we were able to prioritize extensibility by using Coq's module system and minimizing proof rewrites as new components are introduced. As presented in our main findings, we were able to mostly automate proof search using the Hammer library for a given component's functions with regards to proofs of other components' subinvariants.

Kurzfassung

Browser werden auf der Grundlage von Spezifikationen implementiert, die von unterschiedlichen Personengruppen verfasst wurden. Spezifikationen für verschiedene Komponenten des Browsers können aufgrund der Komplexität der Software so widersprüchlich sein, dass Sicherheitslücken entstehen. Eine Möglichkeit, die Wahrscheinlichkeit solcher Spezifikationsfehler zu verringern, besteht darin, ein formales Modell des Browsers zu erstellen, das mechanisch auf Konflikte überprüft werden kann, anstatt sich darauf zu verlassen, dass Menschen informelle schriftliche Beschreibungen auf potenzielle Probleme untersuchen.

Wir schlagen ein neues Browser-Spezifikationsmodell vor, das mithilfe von Verfeinerungstypen überprüft, ob alle bereitgestellten API-Funktionen unsere definierte Sicherheitsinvariante beibehalten. Das Modell ist ein Übergangssystem, in dem der Zustand die Browser-Interna und die Ereignisverfolgung sind und die API-Funktionen Zustandsübergänge sind. Wenn alle API-Funktionen sicher sind, kann jedes Skript, das nur über diese Funktionen mit dem Browser interagiert, die Invariante nicht verletzen. Die Invariante ist eine Konjunktion bestehender Invarianten für Komponentensicherheit aus der Literatur, die wir als Subinvarianten bezeichnen.

Mit diesem Ansatz können wir nachweisen, dass Skripte sicher sind, ohne den Inhalt der Skripte selbst zu modellieren. Wir sind auch in der Lage, ungebundene Spuren zu modellieren, ohne die exponentielle Explosion eines traditionelleren Ansatzes zur Erkundung des Zustandsraums. Zusätzlich dazu, dass wir das Übergangssystem nicht erforscht haben, konnten wir der Erweiterbarkeit Priorität einräumen, indem wir das Modulsystem von Coq nutzten und das Umschreiben von Beweisen bei der Einführung neuer Komponenten minimierten. Wie in unseren Hauptergebnissen dargestellt, konnten wir die Beweissuche mithilfe der Hammer-Bibliothek für die Funktionen einer bestimmten Komponente im Hinblick auf Beweise für die Subinvarianten anderer Komponenten weitgehend automatisieren.

Contents

Abstract **ix**

Kurzfassung **xi**

Contents **xiii**

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Structure of the Thesis . . . 3

2 Background **5**
2.1 Browser Operation . . . 5
2.2 Formal Verification . . . 10
2.3 Information Flow . . . 16

3 Model **19**
3.1 API Functions . . . 19
3.2 Model State . . . 20
3.3 Invariants . . . 21
3.4 Components . . . 22
3.5 Extensibility . . . 30
3.6 What is not modeled . . . 34

4 Evaluation **37**
4.1 Case Studies . . . 37
4.2 Proof Evaluation . . . 38

5 Related Work **41**

6 Conclusion **45**
6.1 Future Work . . . 46

List of Figures **49**

List of Tables **51**

CHAPTER 1

Introduction

We trust applications running in web browsers to handle some of our most personal data–private emails, health records, banking details. While there are many parts of browser security, one crucial aspect is placing the correct restrictions on the arbitrary code that it might run. We need to correctly balance the utility of running complex software on a universal platform with the security of sandboxing that keeps multiple applications safe from each other, all while grappling with the problem that scripts in a Turing-complete language like JavaScript cannot be automatically checked for complex safety properties.

Originally browsers did not have this issue because they only displayed static pages of text, but they have evolved to become the complex platform they are today. A modern browser has a large code base, for example Chrome is estimated to consist of 31m lines of code [Syn24]. Browser specifications are more tractable, but even those can have complex interactions. Unlike a centralized authority like a large tech company, browser specifications are maintained by multiple consortia so that a website's code can be run on any browser on any operating system [HTM]. This unique approach to designing a critical piece of software makes it difficult to prioritize the multiple security goals of different components of the browser. Even though a particular working group might look at other, potentially conflicting, components when updating their specification, they are likely not as familiar with the other components and therefore may struggle to see how potentially ambiguous security goals might be violated.

One way to address this ambiguity is to keep an updated formal model of the specification which can be used to check that specifications do not interfere with each other. Browser developers might still introduce security bugs when implementing the code for a particular browser, but at least they would start from specifications with formal guarantees. The model would also require formal statements of security conditions rather than informal. Once the consortia agree on a model, they could use it to check that changes to their components do not violate the formal security goals of any of the existing components.

Multiple projects have attempted to formalize web specifications to check for security properties. Bohannon and Pierce introduced Featherweight Firefox in 2010, a model of the browser including operational semantics for a formal model of JavaScript [BP10]. The model is a reactive system: a transition system which moves between producing and consuming states based on outside events. It was initially presented as an OCaml model, but the same author later formalized it in Coq, a proof assistant that can check proofs to prevent human error [Boh12]. However, the model does not include modern web components like iFrames or service workers and is difficult to extend because of the complexity of the model, especially of the JavaScript semantics.

A more comprehensive model of the browser is Fett et al.'s Web Infrastructure Model, but it only supports pen-and-paper proofs, which leaves more room for human error [FKS17].

Another approach developed at TU Wien is WebSpec, which uses bounded model checking to explore finite traces in a transition system to check if security invariants are violated [VFB+22]. Exploring concrete traces makes it possible to automatically find concrete attacks that violate invariants, which make the invariant violations possible to recreate in browser implementations. However, because traces are finite, statements about absence of bugs need to be proven manually.

We propose a new model, WebAPISpec, which can mechanically prove security invariants hold for unbounded traces in a transition system representing the browser and is extensible. We handle unbounded traces by proving the security of each API function that a script running in the browser might call. Security is defined by a conjunction of security invariants for all modeled components, which are derived from the literature. Once every API function is proven to maintain the overall invariant, we can then conclude that a transition system that starts in a secure state and only uses secure transitions will always be in a secure state.

We can then use the model to check the proposed implementations of new functions do not violate the security invariant. Multiple candidate implementations, all of which are secure, can be supplied to browser developers after the specifications are finalized.

We use the proof assistant Coq's refinement type checking to enforce this in a way that can be proven mechanically. Refinement types make security proofs of unbound traces feasible and also make our model more easily extensible. If we were to prove the security of the transition system using a method which explores concrete traces, each new component added would increase the search space exponentially because the number of possible transitions at each state would increase. Our method instead increases proof obligations roughly linearly with components modeled since the soundness of each new function is proven in isolation.

Once a function is proven secure the only source of increased proof burden is extending the overall invariant to cover more security conditions. To make this easier to handle we also provide proof techniques that we show minimize handwritten proofs needed to prove a new security condition for a component other than the one the API function is part

of. Finally, we use typical software engineering modularization practices to minimize interference between unrelated components so that the model is easy to extend.

1.1 Contributions

We contribute the following:

1. WebAPISpec, a new model[1] of the browser, which includes core browser operation, cookies, and service workers, with formalized security guarantees. The guarantees are combined into an invariant which can enforce a range of security mechanisms, protecting both data integrity and data confidentiality.

2. A proof technique based on refinement types which verifies the invariant holds for unbound traces of browser behavior.

3. An evaluation of how we were able to use modularity to reduce proof rewriting as the model expanded.

The model enforces useful security guarantees without analyzing script content or requiring websites to make changes. It is well founded because the API functions are checked against the invariant. If an existing API function specification does not obey the invariant we propose alternative functions which do maintain the invariant.

We can check for correct implementation of standard web security notions as well as information flow properties based on scripts that might influence some data, which does not require labeling from the websites themselves.

The model is modular so that new API functions and new components can be added with minimal disruption to existing proofs.

1.2 Structure of the Thesis

The rest of this thesis is structured as follows:

In chapter 2 we present background information. First we review basic browser operation, specifically the components of the browser that we have modeled. We then introduce the approach we took to formal verification in order to enforce the invariant on the implementation of API functions. Finally we present the basics of information flow, which is used in some subinvariants.

In chapter 3 we present our model, first the parts of each component and then the three components which we have implemented: core browser operation, cookies, and service workers. We also present our proof methods in more detail.

[1] Code available at https://github.com/alee122/webapispec.

In chapter 4 we present both qualitative and quantitative analysis of the model. First we present two case studies detailing each of the non-core components. Then we present analysis of proof automation for cross-component proofs; that is, proofs of invariant clauses which represent security conditions of a component other than the one to which the API function being proven belongs.

Finally, chapter 5 explores related work and chapter 6 sums up the results of this thesis and proposes future work.

CHAPTER 2

Background

In this chapter we will review: how browsers work; one approach to formal verification, refinement types; and information flow.

2.1 Browser Operation

Browsers have become more complex over time as they offer more utility to users, so we will present how they work in stages. First we will cover the basics: requesting a specific resource and processing the server response containing it. Second, we will introduce cookies, which allow the browser to identify itself to the server. Third we will introduce service workers, JavaScript programs which act as a proxy for the server so that a browser can handle repeated requests without continuous access to the server.

2.1.1 Core Browser Operation

At its core, the browser provides access to information stored on a remote server and displays that information to the user. In order to request remote information, the browser needs to be able to specify exactly what resource it would like the server to send. Every resource on the web can be specified using a URL (uniform resource locator). In the following definitions we will use an ongoing example, in which we get the static landing page at example.com.

URLs

A URL (Figure 2.1) is a string which uniquely specifies a resource on the web and is composed of the following:

- Scheme: specifies the protocol to be used. In this case HTTP (hypertext transfer protocol) specifies particular headers sent in requests and responses, given an existing

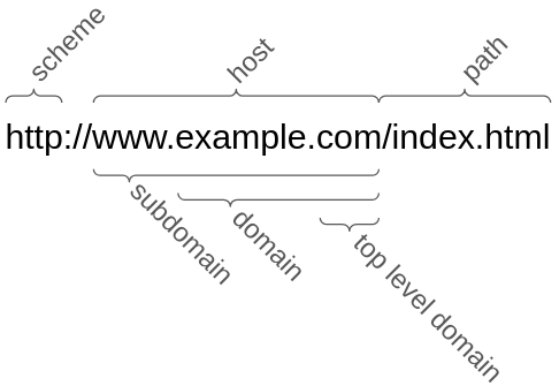

Figure 2.1: URL content.

connection with the server. Other schemes do more than select headers–for instance HTTPS (HTTP-Secure) also requires the browser and server to communicate over an encrypted connection–and some schemes do less–for instance the "file" scheme which displays local files.

- Host: specifies which server to ask for the document. The host is the identity of the server storing the information in a human-readable form. In order to accommodate the vast number of servers the host is broken into the following subsections:
  - Top level domain: a fixed set of suffixes that usually specify the purpose of holder of the domain name. For instance .com for businesses, .gov for official government websites, etc.
  - Domain: identifies the organization or person that holds the information.
  - Subdomain: optional and for internal organization. An organization can have multiple subdomains, and also nest subdomains within each other by adding another prefix string. The most widely known subdomain is probably "www" for "World Wide Web," from a time when an organization might have a website on company servers as an afterthought.
- Path: specifies which resource is requested.

HTTP Requests

Once the browser has established a connection with the server then it is able to send a request. The request above is a minimal example, which contains the following:

- Method: the operation the browser is asking the connected server to perform; in this case let the browser `GET` information.

```
GET /index.html HTTP/1.0
Host: example.com
```

Figure 2.2: HTTP Request header.

- Path: the document the server should send (from the URL path).
- HTTP Version: the required format/headers the browser would like to receive.
- Every subsequent line is an optional pair of a header and value, in this case the host that the request expects to be in contact with.

HTTP Responses

```
HTTP/1.0 200 OK

<!doctype html>...
```

Figure 2.3: HTTP Response header.

The server sends the document, if it exists as specified in the request, with an appropriate header, as in Figure 2.3. It contains the following:

- HTTP Version: as in the request, the response specifies the version of the protocol that it is using.
- Response code: in this case the default success code: 200.
- Response description: here "OK" because the request was successful.
- Response Headers: a set of pairs of headers and values.
- Body: the content of the requested document.

Internal State

The browser stores the body of the responses it receives to display to the user. While originally responses held only text with links to other pages, over time browsers have evolved to handle complex formatting and dynamic pages by storing responses in a tree like structure called the DOM. Formatting is out of scope for this thesis, but dynamic pages fundamentally run using scripts that could come from adversaries who will exploit access to honest pages running in the browser. The structure can also accommodate nested windows, which give parent windows control over their child windows that can potentially be exploited.

Frame Navigation Security

Correct scoping of script influence is crucial to core browser security. For instance, scripts should only be allowed to redirect frames that are a child window of the script, not an unrelated frame [BJM09].

2.1.2 Cookies

The core browser presented is stateless: there is no way for a user to have a unique view of the page, which is inconvenient if a website wants to prove the user the ability to stay logged in, see or a consistent shopping cart, or any other user specific experience. Websites can change this using cookies. A cookie is a small amount of data stored by the browser and added to the request header to customize the experience of the web page. The data actually sent in the request is a key-value pair of strings. When a request is made to an origin that has associated cookies, all those pairs are added to the request headers as shown in Figure 2.4. A server adds a cookie to the browser's stored cookies (the "cookie jar") by sending the details about the cookie in a response header as shown in Figure 2.5.

The server can send more information about the cookie than just the key-value pair that is sent with requests to the server. It can also specify the following attributes in the `SetCookie` response header:

- `Domain`: An optional setting which allows the server to direct the browser to send the cookie in requests to subdomains as well as domains that exactly match the domain which originally sent the `SetCookie`.
- `Path`: A setting which limits where cookies are sent to a path within a website; the default `/` path means the cookie is sent in requests to all locations.
- `Secure`: If specified the cookie can only be sent over an HTTPS connection.
- `HTTPOnly`: If specified the cookie cannot be accessed by the JavaScript cookie API.
- `SameSite`: Can take one of three possible values:
  - `Lax`: The default value, the cookie is sent with requests when they are from the cookie's origin site or when the user navigates to the origin site.
  - `Strict`: The cookie is only sent with requests that come from the cookie's origin site.
  - `None`: The cookie can be sent to cross-site destination, including to sites with different origins; however the `Secure` attribute must be specified.

In addition to the attributes in the `SetCookie`, the name of the cookie limits the use of the cookie. The name might have one of the following prefixes:

- `__Secure`: Requires the cookie to have the `Secure` attribute set.
- `__Host`: Requires the cookie to have the `Secure` attribute set, the `Domain` attribute not set, and the `Path` attribute set to `/`. This way cookies are limited to the `SetCookie`'s sending domain.

If any of the prefix requirements above are not met the cookie is not stored.

```
GET /index.html HTTP/1.0
Host: example.com
Cookie: name1=value1
```

Figure 2.4: HTTP Request header with cookies.

```
HTTP/1.0 200 OK
Set-Cookie: \_\_Securename2=value2; Secure

<!doctype html>...
```

Figure 2.5: HTTP Response header with set cookie that uses a prefix and `Secure` attribute.

Cookie Security

From the browser perspective, cookie security requires enforcing the rules for the `SetCookie` attributes specified by the server. For instance, preventing scripts from accessing a cookie that is marked `HTTPOnly`.

This becomes more challenging as further components are introduced, for instance providing API functions to allow scripts to read raw responses might reveal cookies that the script could not request from the cookie jar.

2.1.3 Service Workers

Finally, the modern browser tries to reduce web content's reliance on continuous server access by providing resources locally if possible, as stated in the Service Worker draft standard [AK22]. In addition to running scripts that make pages interactive, the website developer can write scripts known as service workers that act as a proxy for the server by providing cached versions of documents where possible.

The service worker cache is stored locally in the browser. The scripts can also encode logic that would otherwise be handled on the server, to more fully emulate a continuous connection.

Service Worker Security

A service worker is a powerful component of the browser and so can have a major security impact if compromised. As shown by Squarcina in [SCM21], a service worker can be hijacked through its storage. Since scripts running on the pages of the browser (or malicious scripts included by an attacker) can by default access the service worker storage, the service worker as currently implemented is a vector for escalating attacks.

2.2 Formal Verification

While "formal verification" refers to a wide variety of techniques here we will focus on only what WebAPISpec uses. First we will introduce invariants as a concept and then show how we can check that an invariant holds at compile time using refinement types. Finally, we will lay out why we use Coq as the tool to check that the refinement typing holds.

2.2.1 Invariants

An invariant is a propositional statement about the state of the code that is expected to be true for the entire execution of the program. A good invariant can be an invaluable tool for narrowing down the source of bugs while testing a program. If it accurately captures the requirements of a library then finding an input which triggers a violation of that invariant caused by a particular function is a clear source of a bug. As an example, consider a dynamic array of integers, as shown in Figure 2.6, which should always have at least as much space allocated as it claims to have allocated.

Dynamic arrays are flexibly sized arrays for memory-managed languages built on a generic statically sized array. The structure of the data type consists of the current size of the dynamic array, the actual size of the memory allocated, and the underlying static array. We also include an initialization function, for which a library user can specify the initial capacity of the array and a function to add an element to the end of the array. The add function calls a helper function which copies the current elements into a new, larger, array if the capacity is not large enough to contain another element. The person developing the library should test it to check that it works as intended before it is used elsewhere.

A developer can use a library called "assert" to check certain properties of the data structure while it is run with different tests, as shown in Figure 2.7. The initialization function establishes the invariant and each function checks that the invariant holds at the beginning and end of its implementation code. The developer can then test the code with different inputs and check if and where an input violates an assert. If there are only assertion failures in the beginning of functions the the developer knows that the content of API functions does not introduce unspecified behavior because that test uses a faulty input (which might benefit from some exception throwing) but does not reflect a logical error in the library itself. If the assert at the end of a function fails then the initial one succeeded and the invariant is violated by the content of that function and there is a

```c
typedef struct {
    int size;
    int capacity;
    int arr[];
} darray;


darray* init(int init_size) {
    darray* da = malloc(sizeof(darray));
    int* new_array = malloc(sizeof(int) * init_size);
    da->size = 0;
    da->capacity = init_size;
    da->arr = new_array;
    return da;
}

darray* expand_array(darray* da) {
    int* new_array = malloc(sizeof(int) * da->capacity * 2);
    for (int i = 0; i < da->size; i++) {
        new_array[i] = da->arr[i];
    }
    free(da->arr);
    da->arr = new_array;
    da->capacity*=2;
    return da;
}

darray* add(darray* da, int elem) {
    if (da->size >= da->capacity) {
        da = expand_array(da);
    }
    da->arr[da->size] = elem;
    da->size++;
    return da;
}
```

Figure 2.6: A dynamic array implemented in C.

logic error. However, this requires the developer to find the right input to trigger a bug when the code runs, and can only prove that some inputs are safe.

Assert statements can not be used to prove the absence of bugs overall, for that we need something stronger. We are already able to check code during compilation for typing errors without needing to run specific tests that trigger typing failures. It is possible to generalize that approach to invariants using refinement types. As seen in Figure 2.8 refinement types can be used to check that the final assertions follows from the initial assertion (using a standard math set notation for “one of the set of dynamic arrays which support the invariant” as the input and output types). The developer can then trust that a dynamic array that is created with the initialization function and only manipulated by functions that preserve the invariant will always be in a state the supports the invariant.

```c
// Same darray struct

darray* init(int init_size) {
    assert(init_size >= 0);
    darray* da = malloc(sizeof(darray));
    int* new_array = malloc(sizeof(int) * init_size);
    da->size = 0;
    da->capacity = init_size;
    da->arr = new_array;
    assert(da->size <= da->capacity);
    return da;
}

darray* expand_array(darray* da) {
    assert(da->size <= da->capacity);
    int* new_array = malloc(sizeof(int) * da->capacity * 2);
    for (int i = 0; i < da->size; i++) {
        new_array[i] = da->arr[i];
    }
    free(da->arr);
    da->arr = new_array;
    da->capacity*=2;
    assert(da->size <= da->capacity);
    return da;
}

darray* add(darray* da, int elem) {
    assert(da->size <= da->capacity);
    if (da->size >= da->capacity) {
        da = expand_array(da);
    }
    da->arr[da->size] = elem;
    da->size++;
    assert(da->size <= da->capacity);
    return da;
}
```

Figure 2.7: A dynamic array implemented in C with assertions.

```
// Same darray struct

{ darray* r | r->size <= r->capacity } init({int init_size | 0 <= init_size}) {
    darray* da = malloc(sizeof(darray));
    int* new_array = malloc(sizeof(int) * init_size);
    da->size = 0;
    da->capacity = init_size;
    da->arr = new_array;
    return da;
}

{darray* r | r->size <= r->capacity} expand_array({darray* da | da->size <= da->capacity})  {
    assert(da->size <= da->capacity);
    int* new_array = malloc(sizeof(int) * da->capacity * 2);
    for (int i = 0; i < da->size; i++) {
        new_array[i] = da->arr[i];
    }
    free(da->arr);
    da->arr = new_array;
    da->capacity*=2;
    assert(da->size <= da->capacity);
    return da;
}

{darray* r | r->size <= r->capacity} add({darray* da | da->size <= da->capacity}, int elem) {
    if (da->size >= da->capacity) {
        da = expand_array(da);
    }
    da->arr[da->size] = elem;
    da->size++;
    return da;
}
```

Figure 2.8: A dynamic array implemented in C with refinement types.

There is a downside, however. Using refinement types makes type checking undecidable in general, and we now need to provide a proof that the correct input types will always lead to an output of the stated type.

2.2.2 Proof Assistants

A proof assistant is a software tool that the user interacts with to find formal proofs [Sch09].

We chose our proof methods in order to make proof writing easier and to minimize manual proof changes as the model is extended. We took lessons from DY*, the paper which also inspired our unbounded trace and invariant approach, which implements secure cryptographic protocols with unbounded loops [BBD+21]. That paper builds a framework on a proof assistant called F*, which we initially considered to take advantage of its separate function writing and proof writing as well as its automated proof search

```
Module Type EXAMPLE.
    Paramter T : Type.
    Parameter eq : T -> T -> bool.
End.
```

Figure 2.9: The definition of the type of a module which needs to be given some type and a function that takes two types and returns a boolean.

using an SMT solver. However, when a proof is not solved automatically the error messages from F* are challenging to understand because a SMT solver does not pinpoint the cause of a proof failure as well as a type theory based proof assistant.

Coq

Ultimately we chose to model the browser in Coq, which has several built in advantages and can be extended with libraries to regain some of the automation lost when abandoning SMT solvers. Coq is a proof assistant that is capable of checking that refinement types hold for functions and also has additional benefits:

Gallina Coq has a built in functional language called Gallina which we can use to implement the API functions.

Modularity Coq has a module system which allows the user to structure large developments so that a set of types, functions, and proofs are packaged together [Doc21]. It is also possible to specify a type of a module, so that the writer can specify exactly what type parameters, function types, and proof statements must be provided in any instantiation of that module. For instance, it might require a generic type and then a function which judges boolean equality of two elements of that type as shown in Figure 2.9.

Figure 2.9 should look familiar to people who have used functional programming languages, but in Coq it is possible to go further and specify proofs in the module type as well. Instead of writing documentation stating that the function `eq` must be an equality function, we can enforce it by declaring the standard equality axioms of reflexivity, transitivity, and symmetry in the module type as shown in Figure 2.10. In order to create an instantiation of an instance of `EXAMPLE` the user names a new module of the signature type:

```
Module Ex : EXAMPLE.
    (* types/implementations of the correct type/proofs of axioms *)
End Ex.
```

The line `End Ex.` will not compile until all names types are instantiated, named functions of the correct types are implemented, and all axioms are proven.

```
Module Type EXAMPLE.
    Paramter T : Type.
    Parameter eq : T -> T -> bool.
    Axiom eq_refl :
        forall x : T, eq x x = true.
    Axiom eq_trans :
        forall x y z : T, eq x y = true -> eq y z = true -> eq x z = true.
    Axiom eq_symm :
        forall x y : T, eq x y = true -> eq y x = true.
End.
```

Figure 2.10: The previous module type extended with axioms to make sure the function checks equality.

Once the module compiles it can be imported into other Coq files and used to write types, functions, and proofs. As instantiated above that is all that can be used, not other proofs and functions that exist about the types wrapped in names in the module type.

The module system also allows transparent references to types:

```
Module ExTransparent <: EXAMPLE.
    (* types/implementations of the correct type/proofs of axioms *)
End ExTransparent.
```

This means that, if type `T` is instantiated with the type `nat` (the natural numbers) it is possible to unfold that type in another proof and operate on natural numbers, which can be helpful for some proofs. Otherwise, with an opaque interface, proofs about `nats` that aren't exposed by the interface cannot be used on elements of type `Ex.T`.

Ltac Coq makes it possible to write proof scripts that produce the proofs themselves in a language called L_{tac}, for "tactic language" (but generally styled Ltac). It is also possible to extend it with custom scripts for common proof steps in a particular project.

Hammer In order to simplify proof search we use the library Hammer [CK18]. Certain goals can then be solved immediately by calling the tactic `hammer`, which runs automated theorem provers to try to find a proof of the current goal. This allows us to take advantage of SMT's automated proof search within Coq. If the search is successful the found proof can replace the call to the hammer tactic to prevent repeating the search every time the proof is checked.

Crucially, Coq does not include proof production in its trusted computing base, the code which must be assumed to be correct. Instead the only thing that is assumed correct is a separate proof-checking kernel which is relatively small. The extensions mentioned here, Ltac and Hammer, do not introduce new sources of errors into Coq proofs.

Figure 2.11: A H/L information flow lattice for confidentiality. The arrow can be read as "can flow" if the data label is the back and the location label is the front.

2.3 Information Flow

Now that we are able to enforce properties on code at compile time, we need to define properties that make the code secure. Calling code "secure" can mean different things depending on the circumstances, but often is defined by the system's ability to enforce confidentiality or integrity for the information it handles. Confidentiality means a piece of information can not be read by the wrong party. Integrity means a particular piece of information comes from a particular source and has not been modified. Browser security is challenging because information is frequently exchanged with external servers and local information can be manipulated by code from external sources running within the same program.

One way to specify preservation of confidentiality or integrity is using information flow, which in this case we use for controlling whether information is stored in a particular location. The information and the location each get a label that reflects their security level, and information is only allowed to flow if they are compatible. This can be expressed as a lattice in which there is a relation from the information label to the valid locations label(s). In the following subsections we introduce a few different information flow labeling schemes for checking confidentiality and integrity.

H/L Labeling

We will begin with preservation of confidentiality. In its simplest form information flow concerns two states: high confidentiality (H) and low confidentiality (L). Both data and storage locations are labeled with one of the states. Storage is H if only trusted parties can read it, and data is H if it should only be readf by trusted parties. It follows that H data can only be stored in H storage. L data can be stored in L storage but also H storage, because protecting data that does not need to be protected does not harm security. In this sense we say that data "can flow" from L to H confidentiality, as seen in Figure 2.11.

Integrity information flow for H and L labels is the inverse: H integrity data can be stored in both H and L labeled storage, resulting in the lattice shown in Figure 2.12. The reasoning is that if some storage is expected to have data which may not come from a safe source (labeled L) it is acceptable to store safely sourced data (labeled H) there because it more than meets the requirement of the storage. L labeled data cannot be placed in H labeled storage, because everything it stores should come from a trusted

Figure 2.12: A H/L information flow lattice for integrity.

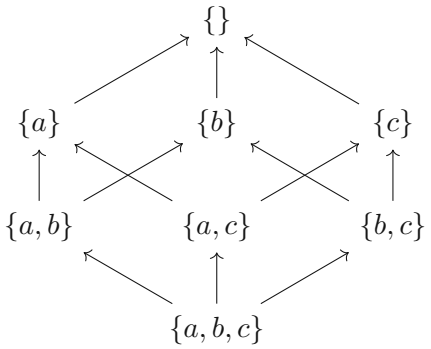

Figure 2.13: A superset information flow lattice for confidentiality. Since it is a lattice the relation is transitive.

source. This relationship is shown in Figure 2.12.

Power Set Labeling

It is possible to use a more complex lattice to encode information flow: the power set lattice. In this case rather than H and L labels for storage and data we label them with an element of the power set of a given set of principals who can access or publish data.

We will again start with confidentiality, in a world in which the existing principals are named a, b, and c. In this case if storage is labeled with a set $\{b, c\}$ then principals b and c can read this storage location so only data labeled with those principals or a superset of them can be stored securely. For example, data labeled $\{a, b, c\}$ could be safely stored because it can be seen by any principal (but in that storage will only be seen by b and c). Data labeled $\{b\}$ could not be stored because it should not be seen by c, which the content of that location might be. This information flow relation forms the superset lattice Figure 2.13.

Integrity information flow is again the inverse. In the same world of principals $\{a, b, c\}$, data can be stored somewhere if its label is a subset of the storage label, resulting in Figure 2.14. Storage s with label $\{b, c\}$ can only have integrity if it accepts data labeled with at most that set of principals. Data labeled $\{b\}$ can be stored without destroying the integrity of s because data edited by just b is acceptable if what you expect is data edited by b and c. Data labeled $\{a, b\}$ does not preserve the integrity of s because a is

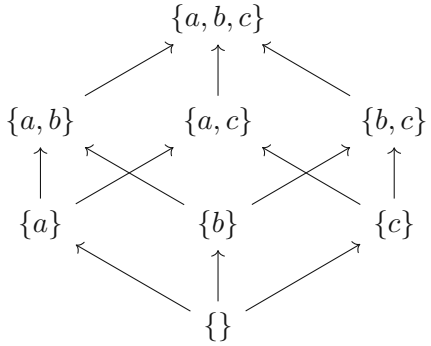

Figure 2.14: A subset information flow lattice for integrity.

not a trusted writer relative to s.

Web Labeling

For the purposes of this thesis we will adapt the power set labeling approach. Since we are interested in the outside resources a website might include we present the following label: a label is a pair $(origin, scripts)$ of the URL which represents the window *origin* and a set *scripts* of the URL origins of all scripts running in the page.

This captures the browsing context in which scripts operate.

CHAPTER 3

Model

WebAPISpec checks the security of browser specifications by modeling them using a transition system. It consists of components which represent logical sections of browser functionality, and which are each broken into the following pieces:

1. The functions of the API;
2. The data structures required by the implemented functions; and
3. The subinvariants which encode the security requirements for that component.

API functions serve as the transitions between different states of the model. By proving that all the API functions preserve the security invariant we can then conclude that, if we start in a safe state, every state in the transition system is safe as shown in Figure 3.1.

The invariant that each API function maintains is explicitly defined as part of the model. The transition system also needs a state to act upon, which includes a history of past events. Each component in the model contributes to all three elements listed above: the functions of its API, representation in the state for the API functions to act upon, and the subinvariants relevant to its own notion of security.

The rest of the chapter is structured as follows: first we introduce the subcomponents, then their implementation details for each component, which are currently the core browser, cookies, and service workers. Finally, we present our methods for keeping the model extensible in terms of modularity and also keeping proofs short and rarely in need of rewriting.

3.1 API Functions

We encode the requirement that the API functions maintain the invariant by giving each API function the following type:

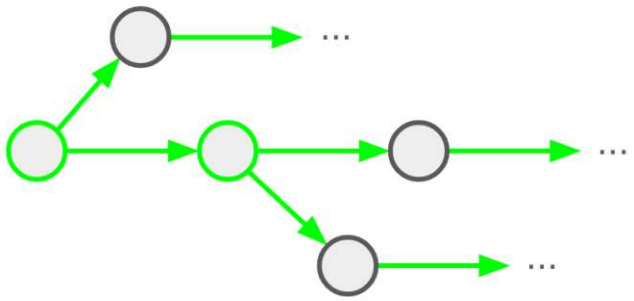

Figure 3.1: If the transition system starts in a safe state and only uses safe transitions then we can conclude each state in the system is also safe.

```
valid_state (* -> other potential inputs *) -> option valid_state
```

in which `valid_state` is an alias for:

```
{s : state | invariant s}
```

That is, the set of states for which the invariant holds.

We can also ignore the "other potential inputs" for now, which gives us the following general type for API functions:

```
{s : state | invariant s} -> option {s : state | invariant s}
```

Restated, this type enforces that if given as an input a state for which the invariant holds the function either returns a state `s'` for which the invariant holds (as `Some s'`) or if that is not possible it returns an error (`None`). Crucially, the function will not return an updated state which violates the invariant. That means that every API function can be composed in any arbitrary order using the option bind operator, [1] and if there is a final state the invariant will hold on that state.

This typing generally does not come for free, as mentioned in 2.2.1, so each API function has a proof that any altered state it returns also fulfills the invariant.

3.2 Model State

The model state consists of a pair (s, t) of the state of the browser and the trace.

[1] An operation which, given two functions `f`, `g` from some type `a -> option a`, checks if `f a` equals `Some a'`; if it does the function returns `g a'`, if not propagates the `None` that was actually returned.

The trace is an unbounded list of all events that have previously occurred. An event is one of the set of all operations that could happen in any component and contains a snapshot of the relevant state information when that operation occurred. For instance, when a response is sent to the browser the trace is extended with a response event that contains the response itself.

The state s is a record of all the data structures for modeled components and a model of the network which consists of an *outbox* and *inbox*. The *outbox* is a queue of all requests that have not been answered by a corresponding response. The *inbox* is an optional response; either there is no response to be processed or there is `Some rp` which corresponds with the top request waiting to be processed in sequence. The outbox queue fixes the order in which responses are handled.

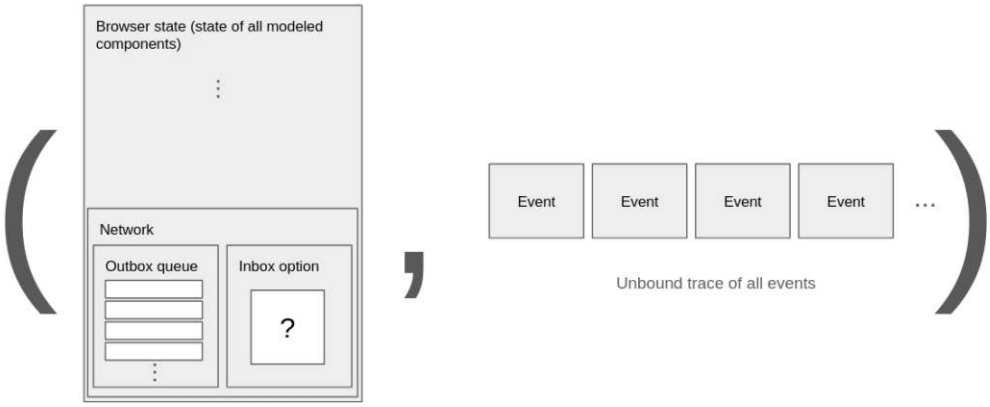

Figure 3.2: Graphic representation of the model.

This is a reactive model, a model which has a neutral state until some kind of event occurs–a new response, a user request–which the browser handles by updating the state or producing output and then returning to a neutral state to wait for the next event [BPS+09].

3.3 Invariants

Invariants are a concise way to encode security requirement as a function from the state of the program to safe (`True`) or unsafe (`False`). They have been used in the web specific security models in the past, for instance in Akhawe's Alloy model [ABL+10]. For more detail refer to subsection 2.2.1.

Each component has a subinvariant or subinvariants: the relevant notion(s) of security. The overall invariant that designates valid states is the conjunction of each modeled component's subinvariant(s). Therefore, each API function must not only enforce

```
Record browser {
    dom : Window ;
    outbox : list UnfilledRequest ;
    inbox : option Response
}
```

Figure 3.3: The browser model data for core operation.

the security relevant to its own component's security, but the security of all modeled components for the conjunction to hold and the function to be typable.

3.3.1 Labels

This is a browser-specific adaption of the idea of subset matrix labels as introduced in section 2.3. In addition to checking that information "can flow" into a certain destination because of the scripts that are running it must also have the same origin in order to flow.

3.4 Components

What follows is descriptions of each of the components represented in the model: the core browser, cookies, and service workers. Each component description includes details about the data structures, events introduced, relevant security conditions, and the API functions implemented.

3.4.1 Core Browser Operation

Data Structures

As shown in Figure 3.2 the state of the model includes the network (*inbox* and *outbox*) as well as the model state. The record that stores the state is shown in Figure 3.3. The core model has state which represent the nested set of windows and elements that might exist within a page. That structure impacts the information stored in the *outbox* so that the model can be properly updated when the *inbox* is populated with the corresponding response. The responses can hold a set of nested elements (but not windows) that need to be fetched in future requests.

Document Object Model The document object model or DOM is the way the browser represents the HTML pages in memory to display them to the user: all nested tags are converted into a tree structure that reflects the structure of the HTML document received.

In our case we are not interested in displaying the page but instead correctly tracking the windows, which delineate the scope of the active scripts for the browsing context. Therefore, we use the inductive data structure shown in Figure 3.4. The goal is to store

```
Inductive Window :=
| WindowFrame : option Url -> option Element -> RequestState -> Window
with Element :=
| Static : Element
| Script : Url -> RequestState -> Element
| iFrame : Url -> RequestState -> Window -> Element
| Pair : Element -> Element -> Element.
```

Figure 3.4: The windows stored in the browser

windows which can contain elements (which can possibly be iFrames that store further nested windows) and track what elements have been successfully loaded from external sources. Elements which require further outside requests have a `RequestState` which represents whether the element is `NotRequested` (yet), `Requested` and waiting for response, or `Loaded` and received in a processed response.

In the simplest case, when a browser is opened it has a blank window. To reflect this our `Window` is initialized as

```
WindowFrame None None NotRequested
```

This means no current URL or `Element` and the `RequestState` of the window is `NotRequested` because there should be nothing in the outbox that will fill this window because there is nothing that belongs in it. If the user then directs the window to a particular URL u (or in other browser initialization scenarios, the browser has stored that particular URL as the homepage or last visited page) the window should reflect that by storing `Some` u. Then the browser, in a search for resources to load, will come across the window with a URL which is `NotRequested` and so add a request for u to the *outbox* paired with a path to the window the response will be stored in, and the window's `RequestState` will become `Requested`. When the corresponding response comes in, the body of that response will be converted into an element as described in section 3.4.1, and the `RequestState` will become `Loaded`.

Elements store many possible kinds of page content. All static elements are abstracted into a single `Static` element. `Scripts` are represented as externally loaded elements which, like windows, can be `NotRequested`, `Requested`, or `Loaded`. `iFrames` store windows within elements, and can independently set the URL a window should be displaying. Finally, the `Pair` element allows us to store an arbitrary number of elements at arbitrary depth.

While this model does not strictly mirror the DOM of a real browser, it does allow us to track the scripts running in a particular browsing context.

Request and Initiator The core `Request` record is shown in Figure 3.5 and contains the URL of the object being requested and the method used to request it, as also

```
Record Request {
    url : Url ;
    method : Method ;
    origin : option Url ;
    body : option nat ;
}
```

Figure 3.5: The request record.

```
Record Response {
    url : Url ;
    body : ResponseContent ;
}
```

Figure 3.6: The response record.

contained in the request header in 2.1.1 albeit in slightly different order and with the protocol stored in the URL. It can also optionally contain the origin, the source URL of the containing window making the request, and a body, here represented as an arbitrary integer.

The design of the DOM means that requests do not come from the browser as a whole but instead come from a particular unloaded resource, which might be arbitrarily nested in the DOM. To make sure we mark the correct element or window as loaded we use `Initiators` as paths through the DOM to the relevant requester. Then we can use that path to mark the correct element loaded (and store any relevant content) once the corresponding response is received.

We call this combination of `Request` and `Initiator` an `UnfilledRequest`, which is what actually gets stored in the *outbox* queue rather than just the request.

Response with Body The response record (Figure 3.6) contains the URL of the resource it is holding and the content itself. The protocol used is stored in the URL and we assume that the server can always return a correct response and so the response code will always be `200 OK`.

The type of the body is the same as the nested `Elements` (not `Windows`) of the DOM (Figure 3.4) without `RequestState` as follows:

- If the body is simply a `Static` element nothing is done to update the DOM, since only the display would change.
- If it is a `Script` then the relevant script (based on the `Initator` in the corresponding `UnfilledRequest`) is marked as `Loaded` when the request is handled.
- Pairs are converted into `Element` pairs.

- iFrames are created with a window based on the containing iFrame response element using the iFrame's URL, no loaded element, and `NotRequested` state.

The converted element can then be stored in the requesting window's element, again based on the request's `Initiator`.

Events The core browser events are:

```
| EvRequest : RequestRecord.unfilled_request -> option url.url -> event
| EvResponse : ResponseRecord.response -> event
| EvJSNavigate : RequestRecord.Initiator -> RequestRecord.Initiator -> event
```

The request event records a event that is registered and the possible origin header, the response event a response, and the navigation event records the location of the script doing the navigating and the location of the frame or window being renavigated.

Relevant Invariant

The core browser invariant enforces the following subinvariant:

```
(Forall (fun cmp => match cmp with
                | (ur__o,(E.EvRequest ur__e o)) =>
                    ur__o = ur__e /\ ur__o.(u_rq).(origin) = o
                | _ => False
                end)
  (zip (outbox b)
     (filter (fun e =>
                match e with
                | E.EvRequest _ _ => true
                | _ => false
                end) t)))
 /\
(length (outbox b) <=
  length ((filter (fun e =>
                     match e with
                     | E.EvRequest _ _ => true
                     | _ => false
                     end) t)))%natall
/\
(Forall (fun event => match event with
                    | E.EvJSNavigate script_loc frame_loc =>
                        RequestRecord.is_parent script_loc frame_loc = true
                    | _ => True
                    end) t)
 /\
(Forall (fun e => match e with
                | (E.EvRequest ur o) => ur.(u_rq).(origin) = o
                | _ => True
                end) t)
```

The conjunction above encodes the following:

- The `UnfilledRequests` all have corresponding request events in the event trace, working backwards in the list and the outbox queue.
- Every JavaScript navigation only occurs for the immediate parent script as presented in [BJM09].
- If a request contains an origin header then that is the actual origin of the requesting parent window [ABL+10].

API Functions

The API functions, which include JavaScript API calls and also core event handling functionality, are:

- `update_dom_with_response`: Consume the response currently in the inbox and the next request to be processed, unpack the contents of the response and store in the DOM tree based on the initiator location stored in the request.
- `make_unloaded_request`: Check the DOM tree for an unloaded resource, for instance a script, and add a new request to the outbox with the location of that unloaded resource.
- `response_to_request`: Given some content as input, populate the inbox with a response that corresponds to the next to process request.
- `handle_js_navigate_frame`: Reset the url of a frame to a new location and set to NotLoaded.
- `handle_js_navigate_window`: Same as above but for a window.

3.4.2 Cookies

Data Structures

Cookies Cookies are key-value pairs of the cookie name and value, which get passed to the server in requests. The name can have different prefixes that limit how the cookie should be used, but otherwise both are arbitrary strings. In Figure 3.7 we represent this with a CookieName that encodes one of three possible prefixes followed by a natural number representing the string, and a value which is a natural number.

SetCookies The browser receives more detailed information about the cookie when it is set by the server, in a structure known as a SetCookie. Set cookies are as shown in Figure 3.8.

```
Record Cookie := {
  cm_name  : CookieName;
  cm_value : nat;
}.
```

Figure 3.7: The request record.

```
Record SetCookie := mkSetCookie {
  sc_name       : CookieName;
  sc_value      : nat;
  sc_domain     : option Domain;
  sc_path       : nat;
  sc_secure     : bool;
  sc_http_only  : bool;
  sc_same_site  : SameSite;
}.
```

Figure 3.8: The request record.

Cookie Jar The Cookie Jar is the name of the cookie store in the browser. It is modeled as a list of all the SetCookies sent to the browser, stored as part of the main `Browser` record.

Impacted Data Structures Cookies and SetCookies were added to the Request and Response data structures, which required some local proof rewriting as discussed in subsection 4.1.1.

Events The events that correspond with cookie manipulation are:

```
| EvJSGetCookie : list SetCookie -> LabelRecord.label -> event
| EvHTTPGetCookie : SetCookie -> event
| EvJSSetCookie : Domain -> SetCookie -> event
| EvHTTPSetCookie : Domain -> list SetCookie -> event
```

The set cookie events reflect updates by JavaScript or from responses with set cookie headers. The HTTP get cookie event adds the right cookies to the latest request. The JavaScript get cookie event checks if the script is able to access cookies.

Relevant Invariant

The cookie subinvariant is as follows:

```
(Forall (fun event =>
    match event with
    | E.EvJSGetCookie scs l =>
        Forall (fun sc =>
            match sc.(CookieRecord.sc_http_only) with
```

```
            | true => LabelRecord.S.Empty l.(LabelRecord.scripts)
            | _ => True
           end) scs
  | _ => True
 end) t).
```

and enforces confidentiality of HttpOnly cookies with regards to scripts as stated in the invariants in [BVV+24].

API Functions

These functions are both JavaScript API endpoint and also key event handling functions.

- `rp_cookie_handler`: Add the `SetCookies` in a response to the cookie jar.
- `user_rq_cookie_handler`: Add cookies to the latest request, if it was made by the user.
- `dom_rq_cookie_handler`: Add cookies to the latest request, if it was made by an unloaded DOM object.
- `js_set_cookie_handler`: Allow a specific script to add a set of cookies passed into the function to the cookie jar.
- `js_get_cookies_handler`: Allow a script to look up all cookie for its origin.
- `js_get_cookie_handler`: Allow a script to request a specific named cookie.

The last two functions do not actually return a SetCookie to a script, since we do not model the script being evaluated, but check if the lookup is acceptable and if not return `None`.

3.4.3 Service Workers

We model the service worker cache with a list of requests and their associated responses, which can either come from the network or be produced by a service worker or script. The service worker itself we do not model explicitly, since our model does not concern itself with the content of scripts, but instead we use a security label as presented in section 2.3 to track the origin page of the service worker and the set of further scripts the service worker has loaded.

Our main security concern with service workers is whether the content of the cache can be contaminated by third party scripts running on the page; that is, the integrity of the cache. We could ask that web pages define a label that represents the required security of the cache, but that would require changes outside of the control of browser developers. Instead we can check integrity relative to the service worker label, so that third party scripts do not inject messages into the cache that the service worker would not produce.

In information flow theoretic terms that means making sure that cache entries were added only by browsing contexts running some subset of the scripts used by the service worker.

Data Structures

The service worker does not introduce new data structures but does alter the browser record to add a cache of request-response pairs and to store a security label representing the current service worker.

Events

```
| EvLoadSW : LabelRecord.label -> event
| EvNewCacheEntry : LabelRecord.label -> RequestRecord.request ->
    ResponseRecord.response -> event
| EvSWRequestResponse : RequestRecord.request -> ResponseRecord.response -> event
| EvSWUpdateCache : RequestRecord.request -> ResponseRecord.response -> event
| EvJSUpdateCache : RequestRecord.request -> ResponseRecord.response -> event
```

Most important is the `NewCacheEntry` event, which captures the security label of a cache entry at the time it is added to the cache.

Relevant Invariant

The subivariant is as follows:

```
(forall rq rp,
       cache_member (rq, rp) b.(browser_cache) ->
       exists l, (corresponding_most_recent_new t rq rp = Some (E.EvNewCacheEntry l rq rp) /\
               LabelRecord.can_flow l b.(sw)))
/\
 (match (most_recent_SW_load t) with
  | Some (E.EvLoadSW l) => LabelRecord.eqb b.(sw) l = true
  | _ => Trueabstracted
  end))
```

and enforces the following safety property on the cache:

- Every item in the cache has a `NewCacheEntry` event in the trace, and the label at that point "can flow" into the current service worker label stored at the browser level.
- The service worker scripts are always equal to those stated in the most recent `update_sw` event.

We feel this is a useful bound to enforce on the cache integrity because we know the service worker must have access to the cache and do not want to allow scripts included outside of it to influence cache entries, based on an attack found by Squarcina [SCM21].

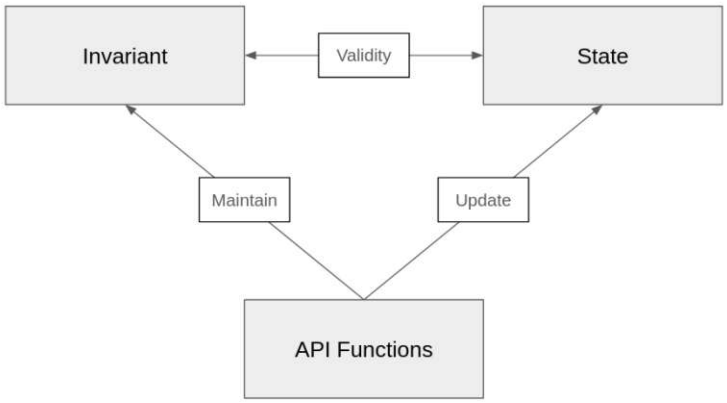

Figure 3.9: The relations between subcomponents of the model.

API Functions

These functions are both JavaScript API endpoint and also key event handling functions.

- `load_and_activate_sw`: Update the service worker of the browser.
- `sw_request_response`: Handle a request using the service worker instead of the server.
- `sw_update_cache`: Update the cache.
- `js_dont_update_cache`: One option for a secure JavaScript cache write access is refusing all access.
- `js_update_cache`: A more useful option is allowing scripts to write to the cache if the scripts doing the writing are a subset of those included in the service worker itself.

Ultimately the model includes all the data structures for each component, the overall invariant is a conjugation of each of the component invariants, and the API functions must enforce that total invariant on the whole state as shown in Figure 3.9.

3.5 Extensibility

We prioritized making the model extensible so that adding a new component requires minimal updates to existing API function proofs. Modularity is supported in three ways: we use type theory not SMT, the data structures that make up the model expose proofs as well as functions, and the files have minimal dependencies.

3.5.1 Type System

The reason we use a type system to enforce security properties rather than automated theorem proving is because theorem proving work can grow exponentially since each additional function adds another possible transition to every state in the transition system. Type checking, conversely, only needs to be checked for each function in isolation. In our case we also benefit from using types to enforce the invariant so that we can prove security of infinite traces in a transition system without explicitly exploring them.

3.5.2 Modularity

Each component has the following three files, in addition to any newly introduced data structures:

1. `<component>EventHandlers.v`: The API functions and the proofs necessary to check typing.

2. `<component>Sigs.v`: The signatures (types) exposed to the API functions. Each can include type declarations, function declarations, and axioms. The types and functions declared must be used in the implementation of the API functions rather than directly manipulating the data structures in case they change later. The same goes for proofs: the proofs of API typing should only use the exposed axioms so that changes to the underlying structure do not disrupt a too specific proof.

   Each component likely has multiple modules: in the case of those modeled each has at least a request module, a response module, and a browser module.

3. `<component>Impl.v`: The modules which implement the signatures declared in `<component>Sigs.v`, including proofs of the required axioms. The types and functions are likely aliases of types and functions in shared raw data types.

When adding a new component it is also necessary to update `Events.v` with new events relevant to the component and to update `Invariant.v` with subinvariants relevant to the component. Existing data structures may also be changed, which might mean updates to other components' `Impl.v` file to maintain the axioms stated in the relevant signature file.

This approach allows us to avoid repeating common proofs, for instance the reflexivity of equality on a structure we defined, in multiple places in the API proofs. We use the module type (discussed in section 2.2.2) to ensure that all the correct proofs exist when we update the underlying data structure.

As a concrete example, when we added cookies to the core request record we had to update the statement of boolean equality because it did not check that the cookies in two records were equal. Before we changed the equality function it was impossible to prove that boolean equality meant two instances of the record were structurally equivalent because

having different cookies only would result in boolean equivalence but not structural equivalence. Because we used the module type system, we caught that mistake before checking the API proofs and only had to update the local proof not every use of it in the API proofs.

Each file is designed with minimal dependencies as shown in 3.10. The content of API

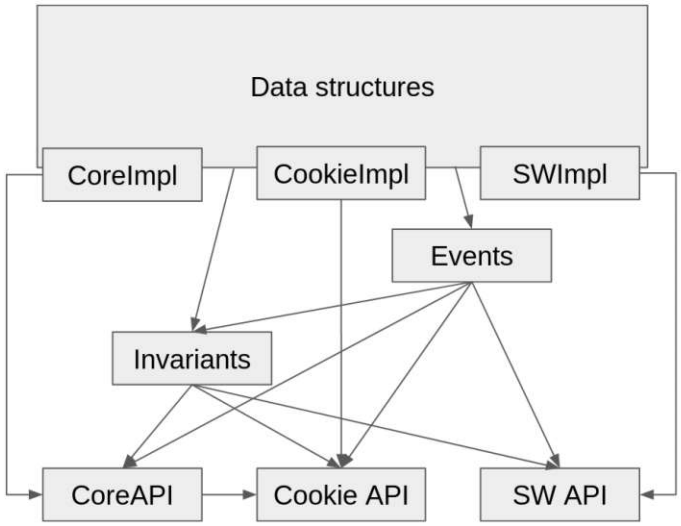

Figure 3.10: File dependencies.

functions is written using an interface to the general data structures so that it is clear exactly what usability is exposed.

We use transparent interfaces because the invariant and events need to have access to all the data structures in the model so that the invariant can enforce the security requirements for all components instead of clashing module implementations of those data structures. Therefore, since the API functions need to refer to the invariant, they cannot use opaque interfaces for the shared data types because the invariant cannot compare the raw types to the interface types. Instead the usage of interfaces rather than raw types in the body of API functions needs to be enforced by hand.

3.5.3 Proof Techniques

In this section we present the structure for all of the API proofs. Figure 3.12 is a representative proof from the Cookie API. When we remove the function-specific details (the body of the function and the non-state inputs) what remains is Figure 3.13, the common skeleton of all proofs with the most straightforward case for automated subproofs (simple application of `sfirstorder`). The last bullet has no `repeat split.` line

```
Ltac prelude :=
    repeat (match goal with
            | [ s : valid |- _ ] => destruct s
            | [ x : state |- _ ] => destruct x
            | [ i : invariant (_, _) |- _ ] => unfold invariant in i
            | [ H : _ /\ _ |- _ ] => destruct H
            | [ H : Forall _ (_ :: _) |- _ ] =>
              eapply Forall_cons_iff in H; destruct H
            end); unfold invariant; split; [idtac | split]; simpl in *.
```

Figure 3.11: The prelude Ltac script.

because there is only a single cookie security invariant. This layout uses several tactics to make it easier to relate a proof to the invariant.

Subset Coercions

As seen in the first line of Figure 3.13, in order to get easy-to-read functions that are similar to the unproven functional version we would write in a language like Haskell, we used the `Program` tactic described in [Soz07]. That tactic allows you to write a standard functional program with standard type inference, and then after presents you with proof obligations that need to be proved with at least some manual interaction.

Ltac

Coq has a built in proof script language called Ltac which decreases the proof burden. Instead of repeating the same standard proof script at the beginning of proving each function we can automate that script as shown in Figure 3.11.

This script breaks up the definitions of existing types until the invariant is not aliased. Then it breaks the invariant into its components in the proof statement, and its conjuncts in the hypotheses.

We also define an unfolding tactic for each of the components which unfolds all the interfaces to reveal the raw definitions that match the definitions used in the invariant.

Line 4 of Figure 3.13, `prelude; unfold_interface`, is written in Ltac, the tactic language for automating proofs in Coq. In this case it is a reference to a custom definition that splits the proof into three subgoals, one for each component invariant. The lines after the single dash bullets are also Coq proof automation, splitting the component invariant into its subgoals the enforces individual security properties.

In this case, each of those subgoals is solved by `!sfirstorder!`, a proof automation tactic from the Hammer library introduced in section 2.2.2.[2]

[2]It would actually be possible to immediately solve the whole proof with `!sfirstorder!`, but that makes it harder to compare automation across proofs and to see which subinvariants might later become unsolvable by the simple automation tactic shown.

```
1  Program Definition js_set_cookie_handler (v : valid)
2    (new_cookies : list C.set_cookie) (i : RQ.initiator) : option valid :=
3    let b := fst v in
4    let t := snd v in
5    match (B.initiator_window_url b i) with
6    | Some u =>
7        let d := u.(host) in
8        match (forallb (fun sc => C.is_valid_setcookie u sc) new_cookies) with
9        | true =>
10           (update_dom_with_response
11              (B.add_to_jar b
12                 (map
13                    (fun rpc => (d, rpc)) new_cookies),
14               (E.EvHTTPSetCookie d new_cookies) :: t))
15       | _ => None
16       end
17   | _ => None
18   end.
19 Next Obligation.
20   prelude; unfold_interface.
21   - repeat split.
22     -- sfirstorder.
23     -- sfirstorder.
24   - repeat split.
25     -- sfirstorder.
26     -- sfirstorder.
27     -- sfirstorder.
28     -- sfirstorder.
29   - sfirstorder.
30 Qed.
```

Figure 3.12: An example API function and proof.

Finally, the last line of the skeleton is `Qed.`, which is a request that the Coq proof kernel check that the proof term is well typed (and so the proof holds) and it matches the theorem statement (it proves what it claims to). This is valuable because it means none of the extensions actually add to the trusted computing base; any errors introduced by extensions will be caught here.

3.6 What is not modeled

Finally, what this model leaves out: the model does not explicitly include the server or the the content of JavaScript files. The first omission is modeled by allowing the reactive system to take any potential response as an input as mentioned in 2.1.1. The second omission is intentional, because analyzing all potential scripts sent to a browser is undecidable. Instead we argue that if the scripts are only allowed to use secure API functions the browser can prevent the scripts from violating the invariant. This means the model covers all possible scripts that impact the browser through API calls (those

```
Program Definition js_set_cookie_handler (v : valid) (* other input *) : option valid :=
  (* function implementation *)
Next Obligation.
  prelude; unfold_interface.
  - repeat split.
    -- sfirstorder.
    -- sfirstorder.
  - repeat split.
    -- sfirstorder.
    -- sfirstorder.
    -- sfirstorder.
    -- sfirstorder.
  - sfirstorder.
Qed.
```

Figure 3.13: The skeleton of all proofs.

that do not make API calls are irrelevant for our security invariants).

CHAPTER 4

Evaluation

For our evaluation we present both qualitative analysis of the process of extending the model with two non-core components, and quantitative analysis of proof burdens relative to subinvariants.

4.1 Case Studies

Starting from the core model, we introduce cookies, which require deep changes to existing data structures but have relatively little proof overhead. Then we introduce service workers, which allow us to demonstrate modularity and the minimality of changes to existing structure of the model. Adding each new component was relatively straightforward, allowing for modifications that strengthen the security invariant.

4.1.1 Cookies

Cookies are interesting because they change the request and response data types used by all components. The use of proof interfaces for all data types meant that only local proof rewrites (things like the reflexivity of equality on a data structure) were required to prove existing API proofs correct.

The only challenge was when writing the functions for the cookie request API: it was not straightforward to make subcalls to the existing core request functions and prove the resulting function enforced all invariants. We had two options

- Restate the content of the existing core request function, but with cookies added from the cookie jar.
- Call the existing core request function within the API function then inject cookies into the new request waiting in the outbox and the new request event in the trace, but that required changing the original statement of the core subinvariant.

We chose the second option because it reduced code rewriting and the invariant change is reasonable.

The original subinvariant required that each `UnfilledRequest` in the outbox have some corresponding event in the event trace. The new subinvariant is stronger, because it requires that the `UnfilledRequests` and corresponding request events in the trace are in the same order not just present somewhere in the event history. This already holds in the transition system, but refinement types do not limit the function to input states produced by the transition system but only to states for which the invariant holds. That includes states for which multiple outbox items are represented by the same event in the trace. In that case injecting cookies into the newest event and newest outbox item might leave a different outbox item without a corresponding request event in the trace. The new ordered outbox invariant eliminates some non-transition system input states from the input state refinement type to make it possible to type check the new cookie injection request function.

4.1.2 Service Workers

The naive approach to enforcing the service worker integrity invariant would be to store a label for every cache entry, but we do not want to impose that overhead. Instead we make sure that every cache entry has at least the integrity of the service worker. First we require that, if a new service worker is loaded that is less permissive (includes fewer scripts in its set) the cache is flushed. That way cache members added when the previous service worker allowed them are not served by the new service worker.

Second, in order to prevent contamination by non-service worker scripts we offer two alternatives to handle scripts attempting to write to the cache:

- Refuse all access to the cache by scripts.
- Allow access to the cache when the set of all scripts running in the browser is a subset of the set of service worker scripts.

We have proven both of them preserve the information flow integrity property on the cache.

Giving two options to enforce the invariant allows browser specification developers to argue that their security invariant is flexible enough to allow different implementation choices instead of being too limiting.

4.2 Proof Evaluation

Our recurring proof skeleton (introduced in Figure 3.13) allows us to usefully compare proof automation between functions. Since we want to minimize manual proving, Table 4.1

Component	Function Name	Core Invariant	Cookie Invariant	SW Invariant
Core	update_dom_with_response	4/13	1/1	2/2
	make_unloaded_request	4/6	1/1	2/2
	respond_to_request	4/4	1/1	2/2
	user_request	4/4	1/1	2/2
	handle_js_navigate_frame	4/8	1/1	2/2
	handle_js_navigate_window	4/8	1/1	2/2
Cookie	rp_cookie_handler	4/4	1/1	2/2
	rq_cookie_insert'	5/12	1/3	2/2
	js_set_cookie_handler	4/4	1/1	2/2
	js_get_cookies_handler	4/4	5/14	2/2
	js_get_cookie_handler	8/8	5/12	4/4
SW	load_and_activate_sw	8/8	2/2	4/4
	sw_request_response	4/4	2/2	1/1
	sw_update_cache	4/4	4/8	1/1
	js_update_cache	4/4	4/9	1/1

Table 4.1: Lines of automated proof to lines of non-skeleton proof, broken up by function.

	General Invariant	Cookie Invariant	SW Invariant
General Proofs	59.3%	100%	100%
Cookie Proofs	86.7%	41.9%	100%
SW Proofs	100%	100%	58.3%

Table 4.2: The percentage of lines of automated proof out of all lines of non-skeleton proof, broken up by component.

is a comparison of Lines of Automated Proof (LoAP) to Lines of Proof (LoP) excluding the skeleton, for each function broken up by component invariant.

As a result we are able to minimize hand written proof requirements, especially proofs of the subinvariants associated with components other than the one to which the API function belonged. Going forward we refer to those subinvariants as other-component-subinvariants rather than proof-component-subinvariants. Table 4.2 shows that most of the proof burden is from proof-component-subinvariants.

We can combine the percentage of LoAP for each component-subinvariant for each component to see how much handwritten proof burden comes from other-component-subinvariants. As shown in Table 4.2, we see that most other-component-subinvariants can be proven with automatic proof search. This is a good sign for our modularity efforts, and shows that future additional components are unlikely to force major proof rewrites on old components.

CHAPTER 5

Related Work

Our work fits into a framework of existing formal methods for web security as laid out in Bugliesi et al.'s survey paper titled *Formal methods for web security* [BCF17]. We also benefit from work on browser attacks when formulating potential security invariants. Finally, our proof approach is inspired by DY*'s approach to unbound traces proven by refinement types [BBD+21].

	Mechanically Checked	Info. Flow Properties	Unbound Traces	Models JS Semantics	Models Cookies	Models SWs
B Browser	○	○	○	○	○	○
Yoshihama Model	○	◐	○	◐	●	○
Featherweight Firefox	●	●	●	●	●	○
Bauer Model	○	○	○	◐	●	○
WebSpec	●	○	○	○	●	●
WebAPISpec	●	◐	●	○	●	●
Alloy	●	○	○	○	●	○
WIM	○	○	○	◐	●	○

Table 5.1: Some related web models assessed for model properties and component inclusion, split into browser models and web platform models.
○ means not included, ◐ partially included, and ● included.

5.0.1 Formal methods for web security

The survey divides formal methods into the following overlapping categories. First, it divides projects between "Security by construction" projects which attempt to provide usable artifacts for a more secure web even if using them would require major changes to the platform as a whole, and "Modelling, verification, and enforcement" works which "formalize and reason about the security of current web technologies" [BCF17]. Our work fits squarely in the second category.

The survey then divides categories by topic: projects are either JavaScript focused, general browser focused, server focused, or focused on modeling the entire platform. Our project most clearly fits into the "general browser focused" category, but in this section we will also make some comparisons to models of the entire platform.

We briefly introduce the most relevant related models in Table 5.1 and in more detail below.

Modeling, verification, and enforcement ∪ The browser view: beyond JavaScript

We will compare WebAPISpec to each of the projects in the most relevant category. Additionally, we will investigate WebSpec [VFB+22], which was not published at the time of the survey but also models the browser beyond JavaScript.

The B browser mode, 2005 The first browser model proposed, this model is I/O automata which supports a UML-like graphic representation [GPS05]. It does not allow for mechanical proof checking but defines automata which represent the browser, secure communication channels, and the user in order to, for instance, "prove the security of password-based user authentication."

Yoshihama's browser model, 2009 A model that provides a big-step semantics so that information flow properties (excluding implicit flows) can be checked [YTTM09]. It is not mechanically checked, but does model the DOM explicitly and check for attacks on nested frames.

Featherweight Firefox, 2010 Featherweight Firefox [BP10] is a model of the browser which explicitly models the content of scripts running in the browser using small-step semantics that approximate a subset of JavaScript. It is similar to our model in that it is a reactive system. While the first model was in OCaml, Bohannon went on to formalize the model in Coq and produce mechanical proofs of the results [Boh12].

However, it does not include modern features like service workers or, crucially, frames that contain nested windows and the difficult browsing context handling that entails. It also does not attempt to solve the problem of proof automation, or of making the model extensible.

Bauer's browser model, 2015 Bauer's model is a transition system with a limited scripting language [BCJ+15]. The model checks information flow policies given explicit labels are provided by websites themselves.

Modelling and reasoning about the DOM, several examples Our representation of the DOM is very limited, and mainly focuses on the way scripts are siloed by iFrames within the DOM. However, several models were mentioned that address the DOM explicitly:

- Minimal DOM, presented by Gardner et al., uses Hoare triples to enforce structural properties on the DOM [GSWZ08].
- Russo et al. presented DOM attacks that can be prevented using runtime information flow control [AMFSR14].
- Rajani et al. presented an information flow control DOM model which does not require explicit labeling from the programmer [RBGH15].

We aim to be more comprehensive.

WebSpec, 2022 This model [VFB+22] was developed at TU Wien, and similarly is a formal model that does not deal with JavaScript files directly. However, it explores finite traces while our invariant based approach can handle unbounded traces.

On the other hand, WebAPISpec does not find concrete attack traces that violates an invariant the way WebSpec does.

Modeling, verification, and enforcement ∪ Formal models for the web platform

These model more of the web than we aim to cover, but are relevant for comparison. We present two of the models in this section.

Formal foundations using Alloy, 2010 One of the earlier models of the web platform, modelled in Alloy, a modelling tool which can check for satisfying models or counter examples [ABL+10].

WIM, 2017 The Web Infrastructure Model [FKS17] is comprehensive, and includes explicitly modeled servers in addition to modeling the contents of scripts. However, only pen-and-paper proofs are possible.

5.0.2 Web Security Invariants

In addition to formal models of the web, we also took inspiration from papers that discuss invariants found while searching for practical attacks on existing browsers. That shows that our platform is able to encode existing useful invariants, rather than only creating invariants that the model can handle.

HttpOnly Cookie Security Bernardo's paper automating the search for browser violations of invariants provided us with our cookie invariant [BVV+24].

Service Worker Cache Data Flow While not formally stated, Squarcina's attack on integrity of the service worker cache inspired our information flow property for the service worker [SCM21].

5.0.3 Proof Approaches

Finally, our approach of proving security of API functions is inspired by a cryptographic protocol analysis paper.

DY⋆ DY⋆ uses refinement types for symbolic analysis of cryptographic protocols [BBD+21]. Like our model, it uses invariants which are maintained by all functions to check that unbounded executions are safe without needing to prove that functions interleave safely.

CHAPTER 6

Conclusion

In this thesis we introduced WebAPISpec, a new model of web browser specifications as a transition system. Uniquely, it allows us to check unbounded traces by using refinement types to check that all API functions maintain our stated invariant, as described in chapter 3. Those function-level proofs allow us to claim that any trace of events starting from a safe browser state (such as the empty state), in which every transition is the application of an API function, can only ever be in a state which satisfies the invariant.

Furthermore, the model is designed to be extended. We presented proof automation tools which minimize hand rewriting of other-component-subinvariants proofs, also in chapter 3. The minimization effort is evaluated quantitatively based on the fraction of hand written proof code of the total code written for each subinvariant in chapter 4.

In the same chapter we report on the experience extending the core browser model using two case studies: cookies and service workers. Cookies are an example of a deep change to the model that touches many of the most important data structures like request and response headers. Service workers are an example of a change to the model that is mostly separate from other components and so has minimal impact on other proofs and simple unrelated subinvariant proofs.

Given the context of the related work presented in chapter 5, we believe WebAPISpec captures a useful subset of browser behavior. Our extensible model is an especially good fit for the way web browser specifications are currently produced: security concerns within a component are rigorously considered by the committee in charge of that component, but problems often arise when a change has an unexpected impact on a different component.

Using our model, a browser specification developer with some mechanical proof experience could check that a new specification upholds her component's security requirements, which she is especially familiar with. If her changes do not lead to any violations of other components' security requirements it will likely be possible to find proofs using automated tools, which means she will not need the same level of knowledge about the

security requirements she is not already familiar with. If there are violations of other components' security properties, WebAPISpec's proof tools help guide the prover to exactly where they occur so that the conflicts can be addressed. As shown in our case studies, those conflicts can be solved by changing the proposed specification or possibly by discussing an invariant change with the other component's specification developers. Once the new changes are proven in the model she can have confidence that the changes do not impact the stated security goals of other components, and so the new specification has a strong case for adoption by browser developers. Furthermore, if strengthening the security of her own component, she can provide multiple proposed implementations that uphold the property in order to argue that the new security standard does not limit the utility of the component too much.

6.1 Future Work

Aside from adding components and extending existing APIs to offer more functions, the model as it exists would benefit from

- a better model of how service workers operate when multiple pages are active, and
- from extending the model to include hyperproperties.

6.1.1 Segmented Cache

Rather than a single service worker for the whole browser each window should have its own service worker. These multiple service workers do not each have access to the whole cache but instead the cache segment of its window origin. That cache segment would have an integrity label that reflected its origin and all the scripts included by service workers in windows that shared that origin.

The difficulty lies in writing inductive proofs over a mutually inductive structure (`Windows/Elements`) that is traversed by an inductive structure (`Initiators`). It should be possible using an explicitly defined induction scheme as follows:

```
Scheme element_mut := Induction for Element Sort Prop
with window_mut := Induction for Window Sort Prop.
```

but we have yet to prove it maintains the invariant for the service worker API functions.

6.1.2 Hyperproperties

Currently our model can handle state properties, propositions which check whether a particular state of the browser internals is safe, and trace properties, propositions which are applied to the state and also the trace of all actions. The next step would be to allow

hyperproperties, propositions which compare multiple traces to see if a property holds [CS10].

A classic example of a hyperproperty is noninterference: in terms of a program with inputs that means that if the low security inputs of two initial states are the same the outcome of the function is the same regardless of the high security inputs.

It is not obvious how these would neatly generalize to our more complex running-script labeling system and to our unbounded traces.

List of Figures

2.1 URL content. 6
2.2 HTTP Request header. 7
2.3 HTTP Response header. 7
2.4 HTTP Request header with cookies. 9
2.5 HTTP Response header with set cookie that uses a prefix and `Secure` attribute. 9
2.6 A dynamic array implemented in C. 11
2.7 A dynamic array implemented in C with assertions. 12
2.8 A dynamic array implemented in C with refinement types. 13
2.9 The definition of the type of a module which needs to be given some type and a function that takes two types and returns a boolean. 14
2.10 The previous module type extended with axioms to make sure the function checks equality. 15
2.11 A H/L information flow lattice for confidentiality. The arrow can be read as "can flow" if the data label is the back and the location label is the front. 16
2.12 A H/L information flow lattice for integrity. 17
2.13 A superset information flow lattice for confidentiality. Since it is a lattice the relation is transitive. 17
2.14 A subset information flow lattice for integrity. 18

3.1 If the transition system starts in a safe state and only uses safe transitions then we can conclude each state in the system is also safe. 20
3.2 Graphic representation of the model. 21
3.3 The browser model data for core operation. 22
3.4 The windows stored in the browser 23
3.5 The request record. 24
3.6 The response record. 24
3.7 The request record. 27
3.8 The request record. 27
3.9 The relations between subcomponents of the model. 30
3.10 File dependencies. 32
3.11 The prelude Ltac script. 33
3.12 An example API function and proof. 34
3.13 The skeleton of all proofs. 35

List of Tables

4.1 Lines of automated proof to lines of non-skeleton proof, broken up by function. 39
4.2 The percentage of lines of automated proof out of all lines of non-skeleton proof, broken up by component. 39

5.1 Some related web models assessed for model properties and component inclusion, split into browser models and web platform models. ○ means not included, ◐ partially included, and ● included. 41

Bibliography

[ABL+10] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 290–304, Edinburgh, United Kingdom, July 2010. IEEE.

[AK22] Jake Archibald and Marijn Kruisselbrink. Service Workers, July 2022. https://www.w3.org/TR/service-workers/.

[AMFSR14] Ana Almeida-Matos, José Fragoso Santos, and Tamara Rezk. An Information Flow Monitor for a Core of DOM: Introducing References and Live Primitives. In Matteo Maffei and Emilio Tuosto, editors, *Trustworthy Global Computing*, volume 8902, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. Series Title: Lecture Notes in Computer Science.

[BBD+21] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. *EuroS&P 2021 - 6th IEEE European Symposium on Security and Privacy*, 2021.

[BCF17] Michele Bugliesi, Stefano Calzavara, and Riccardo Focardi. Formal methods for web security. *Journal of Logical and Algebraic Methods in Programming*, 87:110–126, 2017.

[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 *Proceedings 2015 Network and Distributed System Security Symposium*, San Diego, CA, 2015. Internet Society.

[BJM09] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, June 2009.

[Boh12] Aaron Bohannon. *Foundations of Web Script Security*. PhD thesis, University of Pennsylvania, 2012.

[BP10] Aaron Bohannon and Benjamin C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, page 11, USA, June 2010. USENIX Association.

[BPS+09] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 79–90, Chicago Illinois USA, November 2009. ACM.

[BVV+24] Pedro Bernardo, Lorenzo Veronese, Valentino Dalla Valle, Stefano Calzavara, Marco Squarcina, Pedro Adão, and Matteo Maffei. Web Platform Threats: Automated Detection of Web Security Issues With WPT. *33rd USENIX Security Symposium*, 2024.

[CK18] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning*, 61(1):423–453, June 2018.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, September 2010.

[Doc21] Coq 8.19.0 Documentation. The Module System, 2021. https://coq.inria.fr/doc/V8.19.0/refman/language/core/modules.html.

[FKS17] Daniel Fett, Ralf Küsters, and Guido Schmitz. The Web Infrastructure Model (WIM). 2017.

[GPS05] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Browser Model for Security Analysis of Browser-Based Protocols. In Sabrina de Capitani di Vimercati, Paul Syverson, and Dieter Gollmann, editors, *Computer Security – ESORICS 2005*, pages 489–508, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[GSWZ08] Philippa Gardner, Gareth Smith, Mark Wheelhouse, and Uri Zarfaty. DOM: Towards a Formal Specification. *PLAN-X 2008, Programming Language Technologies for XML, an ACM SIGPLAN Workshop Colocated with POPL 2008*, 2008.

[HTM] HTML5. The Web Platform: Browser technologies. https://web.archive.org/web/20200614193921/https://platform.html5.org/.

[RBGH15] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. volume 2015-September, pages 366–379, 2015. ISSN: 1063-6900.

[Sch09] Patrick Schnider. An Introduction to Proof Assistants, 2009.

[SCM21] Marco Squarcina, Stefano Calzavara, and Matteo Maffei. The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 432–443, May 2021.

[Soz07] Matthieu Sozeau. Subset Coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502, pages 237–252. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science.

[Syn24] Synopsys. The Chromium (Google Chrome) Open Source Project on Open Hub: Languages Page, 2024. https://openhub.net/p/chrome/analyses/latest/languages_summary.

[VFB+22] Lorenzo Veronese, Benjamin Farinier, Pedro Bernardo, Mauro Tempesta, Marco Squarcina, and Matteo Maffei. WebSpec: Towards Machine-Checked Analysis of Browser Security Mechanisms, September 2022.

[YTTM09] Sachiko Yoshihama, Takaaki Tateishi, Naoshi Tabuchi, and Tsutomu Matsumoto. Information-Flow-Based Access Control for Web Browsers. *IEICE Transactions on Information and Systems*, E92-D(5):836–850, 2009.