Informatics

Privatsphärebewahrende Authentifizierte Schlüsselaustauschverfahren

Modellierung, Konstruktionen, Beweise und Verifikation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Andreas Weninger, BSc.
Matrikelnummer 01526989

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Matteo Maffei
Mitwirkung: Dr. Daniel Slamanig

Wien, 15. Dezember 2020

Andreas Weninger | Matteo Maffei

Informatics

Privacy Preserving Authenticated Key Exchange

Modelling, Constructions, Proofs and Formal Verification

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Andreas Weninger, BSc.
Registration Number 01526989

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Matteo Maffei
Assistance: Dr. Daniel Slamanig

Vienna, 15th December, 2020

Andreas Weninger

Matteo Maffei

Erklärung zur Verfassung der Arbeit

Andreas Weninger, 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, 15. Dezember 2020

Andreas Weninger

Danksagung

Ich bedanke mich für die Betreuung und vielen hilfreichen Erklärungen von Prof. Matteo Maffei und Dr. Daniel Slamanig. Außerdem danke ich dem Austrian Institute of Technology (AIT) für das Unterstützen und Finanzieren dieser Arbeit. Weiters danke ich dem AIT sowie konkret Dr. Sebastian Ramacher und nochmal Dr. Daniel Slamanig für die gemeinsame Arbeit an der mit dieser Masterarbeit verbundenen Publikation "Privacy-Preserving Authenticated Key Exchange: Stronger Privacy and Generic Constructions" welche beim 26th European Symposium on Research in Computer Security (ESORICS 2021) akzeptiert wurde und im Tagungsband veröffentlicht wird.

Acknowledgements

I would like to thank Prof. Matteo Maffei and Dr. Daniel Slamanig for supervising this thesis and their many helpful explanations. Furthermore I would like to show my appreciation for the Austrian Institute of Technology (AIT), who supported and funded this work. I also thank the AIT, particularly Dr. Sebastian Ramacher and again Dr. Daniel Slamanig for the successful joint work on the paper "Privacy-Preserving Authenticated Key Exchange: Stronger Privacy and Generic Constructions" which will be presented and published in the proceedings of the 26th European Symposium on Research in Computer Security (ESORICS 2021).

Kurzfassung

Privatsphärebewahrende authentifizierte Schlüsselaustauschverfahren (PPAKE, von engl. Privacy Preserving Authenticated Key Exchange) sind AKE (engl. Authenticated Key Exchange) Protkolle, die so konzipiert werden, dass sie die Identität der beiden Kommunikationspartner vor Dritten geheim halten. PPAKE Protokolle wurden bereits in der Vergangenheit betrachtet. In diesem Werk möchten wir die bestehenden formalen Privatsphäreeigenschaften solcher Protokolle stärken. Der wichtigste Zusatz ist, dass wir auch Angreifer betrachten, die den Protokollablauf nicht korrekt beenden (z.B. weil sie sich nicht authentifizieren können). Auch derartige Angriffe sind relevant, da es dem Angreifer möglicherweise egal ist, ob der Protkollablauf abgebrochen wird, nachdem er die Identität seines Zieles herausgefunden hat. Zusätzlich präsentieren wir ein formales Modell das diese Eigenschaften abbildet und mehrere Protkolle, die unterschiedlich starke Privatsphärebewahrungseigenschaften erfüllen. Eines davon ist eine genersiche Konstruktion aus generischen kryptografischen Grundbausteinen und kann daher auf eine Art instanziiert werden, von der angenommen wird dass sie selbst gegen zukünftige Quantencomputer sicher ist. Zudem präsentieren wir formale Beweise aller Protokolle in dem von uns eingeführten Modell.

Der zweite Teil dieser Masterarbeit behandelt die automatische Verifikation der Privatsphäreeigenschaften des wichtigsten Protokolls aus dem ersten Teil. Automatische Verifikation wird verwendet, um entweder einen Angriff gegen ein Protokoll zu finden, oder festzustellen dass die angegebenen Eigenschaften tatsächlich erüllt sind. Dadurch wird die Wahrscheinlichkeit, in den von Menschenhand geschriebenen Beweisen einen Fehler gemacht zu haben, minimiert. Als erstes untersuchten wir die automatische Verifikationssoftware "Tamarin Prover", die jedoch, bevor der zugeteilte Arbeitsspeicher von ca. 60 GB aufgebraucht war, zu keinem Ergebnis führte (weder einem Beweis noch einem Angriff). Daher nutzten wir stattdessen die Verifikationssoftware ProVerif und konnten die gewünschten Eigenschaften erfolgreich beweisen. In diesem Werk präsentieren wir sowohl unsere Tamarin- als auch unsere ProVerif-Formulierung.

Abstract

Privacy preserving authenticated key exchange (PPAKE) protocols are authenticated key exchange (AKE) protocols that aim to hide the identities of the communicating parties from third parties. Hence the security models of AKE are extended with additional properties. PPAKE protocols have been studied previously. Our aim is to strengthen the existing privacy properties of such protocols. Most notably we additionally consider attacks in which the adversary does not complete the protocol run (e.g. due to the inability to authenticate itself). These attacks are relevant because since some adversaries might not even care if the protocol run is aborted after they deanonymize their target. Furthermore we introduce a formal model that incorporates these properties and several protocols that fulfill different levels of privacy. One of the protocols is a generic construction from generic cryptographic building blocks and hence allows for a post-quantum secure instantiation. Additonally we present formal proofs of all protocols in our model.

The second part of this thesis deals with the automated verification of the privacy properties of the main protocol of the first part. Automated verification is used to either find an attack or conclude that the specified properties indeed hold. This gives additional confidence in the correctness of the security proofs contained in this work. First we evaluated the protocol using the Tamarin Prover, which however is unable to finish its proof or find a contradiction with the given resources (approx. 60 GB memory). Then we utilized the verification software ProVerif and were able to prove the security of the protocol. We will present both the Tamarin Prover encoding as well as the ProVerif encoding.

Note

Part of this work is also used in the joint work “Privacy-Preserving Authenticated Key Exchange: Stronger Privacy and Generic Constructions”[RSW21] by Sebastian Ramacher, Daniel Slamanig and me (Andreas Weninger) that was accepted to ESORICS 2021.

Contents

CHAPTER 1

Introduction

1.1 Motivation

1.1.1 PPAKE

In modern times, communicating over the Internet without encryption is unthinkable. When two users send messages to each other, encryption ensures that no third party is able to read the messages. Recently, there is increased interest in also hiding the identities of the communicating parties. When an adversary intercepts some message, they might be able to deanonymize users through their network routing information (like IP addresses). However, this is not the focus of this work, since there are scenarios in which this is not relevant. On the one hand users may limit the usefulness of such information by using services like Virtual Private Networks (VPN) [FH98] or Tor [DMS04]. On the other hand there are networks in which no such routing information is sent (e.g. in many wireless networks all messages are broadcast). However, even in these scenarios, where network level information is not a problem, adversaries may be able to deanonymize users due to the *content* of the sent messages. We will now motivate how the basic goal of efficient encrypted communication can subsequently lead to such privacy problems.

There are algorithms, such as the advanced encryption standard (AES) that allow two users to exchange messages in a secure way, granted that they both know the secret key (i.e. some information that is only available to these two users). When dealing with a large group of users, sharing this secret among all users implies that all members can read all messages. On the other hand it is infeasible for each user to confidentially exchange secret keys with all other users and to then store them until needed.

Key exchange (KE) protocols, such as the Diffie-Hellman Key Exchange (DHKE)[DH76], remove the need to securely exchange keys with all other users beforehand. Instead of previously establishing a shared secret, users will run the key exchange protocol directly prior to encrypting their messages. Consequently, users do not need to share and maintain keys for all other users, and instead only need to maintain the key that they exchanged with their current communication partner and only for the current session. Furthermore, key exchange protocols provide security even if eavesdroppers are present, whereas sharing a key might otherwise require a secure method.

Alice	**Bob**
Draw random x	
$X = g^x \bmod p$	
$\xrightarrow{\quad X \quad}$	
	Draw random y
	$Y = g^y \bmod p$
	$\xleftarrow{\quad Y \quad}$
$k = Y^x \bmod p$	$k = X^y \bmod p$

Figure 1.1: DHKE

Figure 1.1 shows the Diffie-Hellman Key Exchange. We assume that there are publicly known $p, g \in \mathbb{N}$, so that $2 \leq g < p, p$ is a prime number. Omitting technical details for the sake of brevity, it is guaranteed that both parties compute the same $k = g^{xy} \bmod p$, which can then be used as a key for encryption. Even if some adversary eavesdrops on this communication and hence knows X and Y, it is assumed that there is no efficient way for them to determine k (this is called the Diffie-Hellman assumption).

However, this algorithm is not secure against so called Man-In-The-Middle (MITM) adversaries. These adversaries are not restricted to only listening to the conversation, but are also able to intercept the sent messages and send their own.

As exemplified in Figure 1.2, the MITM adversary simply runs the protocol with both Alice and Bob and thereby arrives at the two keys k' and k^*. Neither Alice nor Bob notice the interference, but instead of communicating with each other they are actually talking to Eve.

In order to fend of such MITM adversaries, the protocol can be extended to use digital signatures (see Figure 1.3). When using digital signature algorithms, each user U has a public key pk_U and a private key sk_U. The user can sign any message, in this example X or Y from the DHKE, by using their private key. All other users are able to then verify that the message was indeed sent by that user by using the public key of that user. Since this means that both users are now authenticated, we speak of an Authenticated Key Exchange (AKE) protocol.

However this approach leads to a new potential problem, namely privacy. While the

Alice	**Eve**	**Bob**
Draw random x $X = g^x \bmod p$		
$\xrightarrow{\quad X \quad}$		
	Draw random y' $Y' = g^{y'}$	
	$\xleftarrow{\quad Y' \quad}$	
	Draw random x^* $X^* = g^{x^*} \bmod p$	
	$\xrightarrow{\quad X^* \quad}$	
		Draw random y $Y = g^y \bmod p$
		$\xleftarrow{\quad Y \quad}$
$k' = (Y')^x$	$k' = X^{(y')}, k^* = Y^{(x^*)}$	$k^* = (X^*)^y$

Figure 1.2: DHKE with MITM adversary (omitted mod p in the final line for brevity)

DHKE does not reveal who is talking to whom, the addition of signatures allows any eavesdropper to determine the identities of the two users. At first glance, authentication and privacy seem to be two incompatible goals. One requires the identity to be known and one seeks to prevent this. However, authentication actually only requires the identity to be known *by the other communication partner*. Indeed, let us consider Figure 1.4.

Figure 1.4 shows a protocol that first runs a DHKE and only afterwards exchanges identity related information (i.e. the signatures) in encrypted messages ($E_k(m)$ denotes the encryption of m with key k). This means that an eavesdropper will not be able to efficiently determine the identities of the communicating parties. At the same time, due to the signatures, the key k is secure even against MITM adversaries (since the users abort and potentially restart the protocol run if a signature is not sent or invalid). Protocols like this are called Privacy Preserving Authenticated Key Exchange (PPAKE) protocols[1].

Note however, that in the aforementioned example (Figure 1.4), the identities of the

[1] This term was introduced in [SSL20] and will be used hereafter.

Alice		Bob
$X = g^x \bmod p$		
$\sigma_{\texttt{Alice}} = \mathsf{Sign}_{\mathsf{sk}_{\mathsf{Alice}}}(X)$		
$\xrightarrow{X, \sigma_{\texttt{Alice}}}$		
		$Y = g^y \bmod p$
		$\sigma_{\texttt{Bob}} = \mathsf{Sign}_{\mathsf{sk}_{\mathsf{Bob}}}(Y)$
		$\xleftarrow{Y, \sigma_{\texttt{Bob}}}$
$k = Y^x \bmod p$		$k = X^y \bmod p$

Figure 1.3: Signed DHKE

Alice		Bob
$\xrightarrow{g^x \bmod p}$		
		$\xleftarrow{g^y \bmod p}$
$k = g^{xy} \bmod p$		$k = g^{xy} \bmod p$
$\xrightarrow{E_k(\sigma_A)}$		
		$\xleftarrow{E_k(\sigma_B)}$

Figure 1.4: Simplified PPAKE

users are not safe against a MITM adversary. The adversary can act as the responder, correctly run the DHKE and receive the third message, without ever needing to send a signature on their own. This problem might at first also seem inherent to PPAKE protocols, since either the initiator or the responder has to "go first" in authenticating themselves. However, later in this work (c.f. Section 3.1) we will present protocols that are able to protect the privacy of both parties even in presence of a MITM adversary.

Previous Work

Privacy preserving authenticated key exchange (PPAKE) protocols have been studied previously. As mentioned before, these protocols fulfill everything that is required from an AKE protocol while also mitigating the risk of identity information being leaked. In the literature, there are many different formulations of these privacy preserving properties. The list below gives a few examples.

1. To the best of our knowledge, the first work that explicitly deals with privacy in the AKE setting is Aiello et al. [ABB+04]. The proposed protocols are designed to only protect the privacy of one party, either the initiator or the responder, against active adversaries. One of the proposed protocols does however protect the privacy of both parties against passive eavesdroppers.

   Aiello et al. reference the paper of Canetti and Krawczyk [CK02], which contains an even earlier mention of "identity concealment". It informally discusses how achieving this notion is possible by encrypting the identities.

2. In [ABF+19], the unilateral authentication in the Transport Layer Security protocol version 1.3 (TLS 1.3) and hence unilateral privacy is investigated. The nature of unilateral authentication strongly limits the possible privacy guarantees. In the TLS 1.3 setting, any client should be allowed to contact a server. Simply consider an active adversary that takes the role of the (unauthenticated) client and runs the protocol normally. The server will authenticate itself and hence reveal its identity.

3. In the work of Schäge, Schwenk and Lauer [SSL20], the Internet Key Exchange (IKEv2) protocol [KHN+14] is examined. The proposed security model guarantees the privacy of both parties if the protocol is completed successfully. This work also coined the term PPAKE.

4. In [Zha16] a construction named CAKE is presented. Again, the security model guarantees the privacy of both parties if the protocol is completed successfully.

This thesis aims to strengthen the existing PPAKE models in the literature by providing privacy guarantees for both the initiator and the responder, in particular even in cases in which the adversary is not able to complete the protocol session successfully. These cases are relevant since some adversaries might not even care if the protocol run is aborted after they deanonymize their target.

1.1.2 Automated Verification

In cryptography, just like in many other areas of science, scientists support their claims with mathematical proofs. The problem one deals with is that of course these proofs could contain mistakes and might thus be incorrect. In order to mitigate this risk, formal verification (also called automated verification) can be used. This entails writing the claims in computer readable form (i.e. some tool-specific input format) and running a formal verification software. Usually the software will continuously apply some calculus, i.e. a set of rules to rewrite the expressions that were provided by the user or derive new knowledge. It terminates upon finding a contradiction or determining that no contradiction was found and no more rules can be applied (which means that the statement is correct). There is also the possibility that the proof search does not terminate. This is unavoidable in any proof system that allows sufficiently sophisticated expressions (e.g. general first order logic statements), since first order logic and similar systems are undecidable, as shown by Gödel's incompleteness theorems.

In this work we utilize and evaluate two tools for formal verification:

Tamarin Prover. The Tamarin Prover is a formal verification software tailored specifically to prove properties of cryptographic protocols. It uses a specific format called *rules* to encode the individual steps of a protocol and allows the user to specify the properties in (a fraction of) first order logic. The state of a system at a point in time during the execution of the protocol is encoded as a multiset of *facts*. A more detailed overview is given in Section 1.3.

ProVerif. ProVerif is another tool for formal verification of cryptographic protocols. It specifies protocols in *process* format. Security properties are encoded as *queries*, which are logical statements of a specific form. There are no state facts as with the Tamarin Prover, instead the order of executing operations is specified since the protocols are encoded in the process format. A more detailed overview is given in Section 1.4.

Verifying a protocol using automated verification has its limits. As mentioned before the tool might not terminate. Even if it does terminate by concluding that the specified properties hold, there might be security flaws in the protocol. The reason is that these tools do not create the type of proofs that are used in the cryptographic literature. There, usually all security properties are derived from some assumption, that a specific mathematical problem is computationally infeasible to solve by the adversary. Tamarin and ProVerif however will only prove that the methods that were defined in the input cannot be used to attack the user-defined property. This means that these tools implicitly assume that there are no relevant mathematical properties aside from those that were specified by the user. To illustrate this, consider some encryption scheme that simply multiplies the message with the key, where the key is always an odd number. Clearly the ciphertext would leak information about the message. If the the ciphertext is an even number, it can be deduced that the message was even too. However the designer of the algorithm might encode the protocol and what it means to multiply and see a "proof successful" message. The reason is that Tamarin and ProVerif do not know about odd or even numbers, unless the user specifies corresponding functions. Since numbers being odd or even are irrelevant for the algorithm itself, the algorithm designer would most likely not include such a definition in the encoding.
However, Tamarin and ProVerif do usually give certainty if they find an attack.[2] These

[2]There are rare cases, in which the tools apply some internal simplifications which lead to wrong

attacks are then described in detail in the tools' outputs. Since they work with the limited tools given to them, they also work in a real-world setting.

This work encodes the protocol Π_{Gen}, which is presented in Section 3.1, and evaluates the Tamarin Prover for this use case. Since that Tamarin encoding cannot be efficiently solved as discussed in Chapter 4, we also show a ProVerif encoding. That encoding is successfully proven.

1.2 Cryptographic Preliminaries

1.2.1 Notation

Security Parameter λ. In this work we deal with asymptotic security. This means that by increasing λ, which is the size of security relevant information (e.g. the length of keys in the system), the workload of an adversary that tries to break our security goals should increase faster than any polynomial (e.g. exponentially). All complexity measurements are hence given as a function of λ.

Negligible Functions. A negligible function is a function $f : \mathbb{N} \to \mathbb{R}$ s.t. for every positive polynomial $\mathsf{poly}(\cdot)$ there is some $N_{\mathsf{poly}} > 0$ s.t. for all $n > N_{\mathsf{poly}}$ it holds that

$$|f(n)| < \frac{1}{\mathsf{poly}(n)}$$

Adversary Advantage. In many security experiments, one may trivially break the security of any protocol by random guessing with at least some probability. For example, if the adversary's goal in an experiment is to determine whether a secret bit is 0 or 1, random guessing will yield a winning probability of $\frac{1}{2}$. The advantage of an adversary denotes how much better the adversary is compared to random guessing. If some adversary in the previous example outputs the correct bit with a probability of $\frac{3}{4}$, this means it has an advantage of $\frac{3}{4} - \frac{1}{2} = \frac{1}{4}$.

Probablistic Polynomial Time (PPT) Adversaries. As mentioned before, breaking the security of our protocols should take a superpolynomial amount of time (in the security parameter λ). Hence all Probablistic Polynomial Time (PPT) adversaries, i.e. adversaries which execute an algorithm that has at most polynomial complexity and may use randomness, should have at most a negligible advantage.

Allowing Adversary $\mathcal{A}$ to Use Oracle O. By $\mathcal{A}^O$ we denote that the adversary $\mathcal{A}$, while running their algorithm, is allowed to flexibly use the oracle O any number of times.

results, namely invalid attacks (c.f. [BBS]). This can be ruled out by manually checking the resulting attack descriptions or setting the option to disable these simplifications.

Algorithm Syntax. In algorithms we write $a \leftarrow b$ to denote that the value b is assigned to the variable a. $a \xleftarrow{R} M$ denotes that the variable a is assigned a uniformly random element from the set M. $||x||$ denotes the length of x (in binary notation).

Group Theory. When writing $\mathcal{G} = (\mathbb{G}, q, g)$ we refer to a cyclic group with the elements in $\mathbb{G}$, the order (i.e. number of elements) q and some generator element g.

1.2.2 Cryptographic Hash Functions

A cryptographic hash function h is an algorithm that maps data of arbitrary size to a fixed size bitstring. It is a one-way function, i.e. any PPT adversary should have only negligible advantage for finding some input value x when given the output value $h(x)$. A hash function is called collision resistant if it is infeasible for a PPT adversary to find a collision, i.e. two different values m_1, m_2 s.t. $h(m_1) = h(m_2)$.

The random oracle model is an idealization of real-world hash functions used in security proofs (c.f. [BR93]). The random oracle (RO), denoted H, is a truly random function. Both the protocols as well as the adversary can query the RO. Consider the setting in which the adversary intercepts a message by the protocol, that contains $y = H(x)$ for some secret x. Since the output of H is random, the adversary does not directly learn anything about x. However, the adversary can try different guesses for x and query $H(x)$ themselves. Furthermore, if the protocol uses the same value x multiple times for different messages, the adversary will notice that y is the same.

As a generally accepted practice, cryptographic proofs may "program" the RO. This means that if the RO is queried for any input for the first time, the proof can make the RO output a specific value depending on the input.

In a practical protocol a RO is then instantiated with a secure hash function, e.g. SHA-2 or SHA-3.

1.2.3 Unauthenticated Two-Move Key Exchange

We denote by Γ a two-move key exchange protocol between two PPT algorithms A and B running in three steps: $(\mathsf{st}_A, \mathsf{out}_A) \leftarrow \Gamma_A^{(1)}(1^\lambda)$ produces A's state and output; $(k_B, \mathsf{out}_B) \leftarrow \Gamma_B^{(1)}(1^\lambda, \mathsf{out}_A)$ on input A's output produces a key $k_B \in \mathcal{K}$ and finally on input B's output $k_A \leftarrow \Gamma_A^{(2)}(1^\lambda, \mathsf{out}_B, \mathsf{st}_A)$ produces a key $k_A \in \mathcal{K}$. Note that $\Gamma_A^{(1)}(\cdot)$ and $\Gamma_B^{(1)}(\cdot,\cdot)$ are stateless functions that only take the specified inputs. Specifically, they do not have access to any long-term keys. Correctness requires that for all $\lambda \in \mathbb{N}$, where λ denotes the security parameter, and all random tapes of A and B we have that $k_A = k_B$, except for a negligible error probability. We use the shorthand $(k, \mathtt{t}rans) \leftarrow \Gamma_{A,B}(1^\lambda)$ to denote a run of the protocol where $\mathtt{t}rans = (\mathsf{out}_A, \mathsf{out}_B)$. We say that a two-move key exchange protocol is secure against eavesdroppers if and only if any PPT adversary $\mathcal{A}$ has only negligible advantage in the following security experiment.

Exp. $\mathsf{Exp}^{\mathsf{eav}}_{\Gamma,\mathcal{A}}(\lambda)$

$k_0 \leftarrow\$ \mathcal{K}$
$b \leftarrow\$ \{0,1\}$
$(k_1, trans) \leftarrow \Gamma(1^\lambda)$
$b' \leftarrow \mathcal{A}(k_b, trans)$
if $b = b'$ then **return** 1 **else return** 0

Figure 1.5: The EAV experiment for a two-move key exchange protocol Γ.

Definition 1. *For any PPT adversary $\mathcal{A}$ the advantage function*

$$\mathsf{Adv}^{\mathsf{eav}}_{\Gamma,\mathcal{A}}(\lambda) := \left|\Pr\left[\mathsf{Exp}^{\mathsf{eav}}_{\Gamma,\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right|,$$

is negligible in λ, where the experiment $\mathsf{Exp}^{\mathsf{eav}}_{\Gamma,\mathcal{A}}(\lambda)$ is given in Figure 1.5 and Γ is a two-move key exchange protocol as above.

For brevity, we will call Γ secure if it is EAV-secure and we will write $\mathsf{out}_A \leftarrow \Gamma(0)$ for A's first message and $\mathsf{out}_B \leftarrow \Gamma(1, \mathsf{out}_A)$ for B's message and denote with $\Gamma.key$ the resulting key if everything is clear from the context.

Lemma 1. *Let Γ be an EAV-secure two-move key exchange protocol Γ, then it holds that:*

$$\Pr[\mathsf{out}_A = \mathsf{out}'_A] \leq \mathsf{negl}(\lambda) \quad \text{and} \quad \Pr[\mathsf{out}_B = \mathsf{out}'_B] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{out}_A, \mathsf{out}'_A$ and $\mathsf{out}_B, \mathsf{out}'_B$ are results of independent calls to $\Gamma(0)$ and $\Gamma(1, \mathsf{out}_A)$, respectively.

Proof. Note that correctness demands that a specific transcript fully determines a single key. If one pair $(\mathsf{out}_A, \mathsf{out}_B)$ could be produced in multiple runs with different resulting keys, then A and B would have no way to tell which key to agree on in a specific run, since they have no common information besides the transcript.
Assume the lemma does not hold, and view the case that $\Pr[\mathsf{out}_A = \mathsf{out}'_A]$ is non-negligible (the other case can be treated analogously). Construct an adversary $\mathcal{A}$ against the security of Γ as follows:

1. Upon receiving $k, (\mathsf{out}_A, \mathsf{out}_B)$, simply call $(\mathsf{st}'_A, \mathsf{out}'_A) \leftarrow \Gamma^{(1)}_A(1^\lambda)$.
2. **Case 1.** $\mathsf{out}'_A \neq \mathsf{out}_A$.
   Output a random bit b'.
3. **Case 2.** $\mathsf{out}'_A = \mathsf{out}_A$.
   Call $k_A \leftarrow \Gamma^{(2)}_A(1^\lambda, \mathsf{out}_B, \mathsf{st}'_A)$. As discussed before, k_A must be identical to the actual key that was derived in the challenger's protocol run. Hence output b' according to whether k_A equals k.

Exp. $\mathsf{Exp}^{\mathsf{se\text{-}ind\text{-}cca}}_{\Omega,\mathcal{A}}(\lambda)$

$k \leftarrow\!\$\, \mathcal{K}$
$(M_0, M_1, l) \leftarrow \mathcal{A}^{E_k,D_k}(\mathsf{pk})$
$b \leftarrow\!\$\, \{0,1\}$
$\mathsf{ctxt}^* \leftarrow E_k(M_b, l)$
$b' \leftarrow \mathcal{A}^{E_k,D_k}(\mathsf{ctxt}^*)$
if $b = b'$ then **return** 1 **else return** 0

Figure 1.6: LH-SE-IND-CCA security for SE Ω.

Since Case 2 has non-negligible probability of happening, this gives $\mathcal{A}$ a non-negligible advantage. □

1.2.4 Symmetric Encryption

A symmetric encryption with padding (SE) scheme Ω with key space $\mathcal{K}$ and message space $\mathcal{M}$ consists of the PPT algorithms (E, D) defined as follows:

$E_k(M,l)$: On input secret key k, message $M \in \mathcal{M}$ and length l ($l \geq |M|$), outputs a ciphertext ctxt.

$D_k(\mathsf{ctxt})$: On input secret key k and ctxt, outputs $M \in \mathcal{M} \cup \{\perp\}$.

A SE Ω is correct if for all $k \leftarrow\!\$\, \mathcal{K}, M \leftarrow\!\$\, \mathcal{M}, l \geq |M|$ it holds that

$$\Pr[\mathsf{ctxt} \leftarrow E_k(M,l) : D_k(\mathsf{ctxt}) = M] = 1.$$

We say a SE Ω is LH-SE-IND-CCA-secure (length-hiding indistinguishable under chosen ciphertext attacks) if and only if any PPT adversary $\mathcal{A}$ has only negligible advantage in the following security experiment. $\mathcal{A}$ outputs messages (M_0, M_1) and length l with $l \geq \max\{M_0, M_1\}$ and, in return, gets $\mathsf{ctxt}^* \leftarrow E_k(M_b, l)$, for $b \leftarrow\!\$\, \{0,1\}$. Eventually, $\mathcal{A}$ outputs a guess b'. If $b = b'$, then the experiment outputs 1. During the experiment $\mathcal{A}$ has access to an encryption oracle E_k and decryption oracle D_k where the adversary can query decryptions of ciphertexts distinct from ctxt^*.

Definition 2. *For any PPT adversary $\mathcal{A}$ the advantage function*

$$\mathsf{Adv}^{\mathsf{se\text{-}ind\text{-}cca}}_{\Omega,\mathcal{A}}(\lambda) := \left|\Pr\left[\mathsf{Exp}^{\mathsf{se\text{-}ind\text{-}cca}}_{\Omega,\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right|,$$

is negligible in λ, where the experiment $\mathsf{Exp}^{\mathsf{se\text{-}ind\text{-}cca}}_{\Omega,\mathcal{A}}(\lambda)$ is given in Figure 1.6 and Ω is a SE as above.

In our protocols we write $E_k(M)$ instead of $E_k(M,l)$ if we assume there to be a suitable publicly known maximum length l.

1.2.5 Public Key Encryption

We briefly recall the definition and security notions of public-key encryption (PKE) including the notion of key-privacy introduced by Bellare et al [BBDP01]. A public-key encryption scheme PKE with message space $\mathcal{M}$ consists of the three PPT algorithms $(\mathsf{PSetup}, \mathsf{PGen}, \mathsf{PEnc}, \mathsf{PDec})$ defined as follows:

$\mathsf{PSetup}(\lambda)$: On input security parameter λ, outputs public parameters pp.

$\mathsf{PGen}(\mathsf{pp})$: On input public parameters pp, outputs public and secret keys $(\mathsf{pk}, \mathsf{sk})$.

$\mathsf{PEnc}_{\mathsf{pk}}(M)$: On input pk and message $M \in \mathcal{M}$, outputs a ciphertext ctxt.

$\mathsf{PDec}_{\mathsf{sk}}(\mathsf{ctxt})$: On input sk and ctxt, outputs $M \in \mathcal{M} \cup \{\perp\}$.

We note that we make the generation of shared public parameters, e.g., the choice of groups, explicit as separate algorithm PSetup. This is necessary for key privacy that we will discuss below.

A PKE scheme is correct if for all $\mathsf{pp} \leftarrow \mathsf{PSetup}(\lambda)$ and $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PGen}(\mathsf{pp})$, then

$$\Pr[c \leftarrow \mathsf{PEnc}_{\mathsf{pk}}(M) : \mathsf{PDec}_{\mathsf{sk}}(c) \neq M] \leq \mathsf{negl}(\lambda).$$

We say a PKE is PKE-IND-CCA-secure if and only if any PPT adversary $\mathcal{A}$ has only negligible advantage in the following security experiment. First, $\mathcal{A}$ gets an honestly generated public key pk. $\mathcal{A}$ outputs equal-length messages (M_0, M_1) and, in return, gets $\mathsf{ctxt}_b^* \leftarrow \mathsf{PEnc}_{\mathsf{pk}}(M_b)$, for $b \leftarrow\!\$\, \{0,1\}$. Eventually, $\mathcal{A}$ outputs a guess b'. If $b = b'$, then the experiment outputs 1. During the experiment $\mathcal{A}$ has access to a decryption oracle $\mathsf{PDec}_{\mathsf{sk}}$ where the adversary can query decryptions of ciphertexts distinct from ctxt^*. If the adversary is not given access to the decryption oracle, then the scheme is PKE-IND-CPA-secure.

Definition 3. *For any PPT adversary $\mathcal{A}$ the advantage function*

$$\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathsf{pke\text{-}ind\text{-}cca}}(\lambda) := \left|\Pr\left[\mathsf{Exp}_{\mathsf{PKE},\mathcal{A}}^{\mathsf{pke\text{-}ind\text{-}cca}}(\lambda) = 1\right] - \frac{1}{2}\right|,$$

is negligible in λ, where the experiment $\mathsf{Exp}_{\mathsf{PKE},\mathcal{A}}^{\mathsf{pke\text{-}ind\text{-}cca}}(\lambda)$ is given in Figure 1.7 and PKE *is a PKE as above.*

We say a PKE PKE is PKE-IK-CCA-secure (also called key private) if and only if any PPT adversary $\mathcal{A}$ has only negligible advantage in the following security experiment. First, $\mathcal{A}$ gets two honestly generated public keys $\mathsf{pk}_0, \mathsf{pk}_1$. $\mathcal{A}$ outputs a message M and, in return, gets $\mathsf{ctxt}_b^* \leftarrow \mathsf{PEnc}_{\mathsf{pk}_b}(M)$, for $b \leftarrow\!\$\, \{0,1\}$. Eventually, $\mathcal{A}$ outputs a guess b'. If $b = b'$, then the experiment outputs 1. During the experiment $\mathcal{A}$ has access to a decryption oracles $\mathsf{PDec}_{\mathsf{sk}_0}$ and $\mathsf{PDec}_{\mathsf{sk}_1}$ where the adversary can query decryptions of ciphertexts distinct from ctxt^*.

Exp. $\mathsf{Exp}^{\mathsf{pke\text{-}ind\text{-}cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda)$

$\mathsf{pp} \leftarrow \mathsf{PSetup}(\lambda)$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PGen}(\mathsf{pp})$
$(M_0, M_1) \leftarrow \mathcal{A}^{\mathsf{PDec}_{\mathsf{sk}}}(\mathsf{pk})$
$b \leftarrow\$ \{0, 1\}$
$\mathsf{ctxt}^* \leftarrow \mathsf{PEnc}_{\mathsf{pk}}(M_b)$
$b' \leftarrow \mathcal{A}^{\mathsf{PDec}_{\mathsf{sk}}}(\mathsf{ctxt}^*)$
if $b = b'$ then **return** 1 **else return** 0

Figure 1.7: PKE-IND-CCA security for PKE PKE.

Exp. $\mathsf{Exp}^{\mathsf{pke\text{-}ik\text{-}cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda)$

$\mathsf{pp} \leftarrow \mathsf{PSetup}(\lambda)$
$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{PGen}(\mathsf{pp}), (\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PGen}(\mathsf{pp})$
$M \leftarrow \mathcal{A}^{\mathsf{PDec}_{\mathsf{sk}_0}, \mathsf{PDec}_{\mathsf{sk}_1}}(\mathsf{pk})$
$b \leftarrow\$ \{0, 1\}$
$\mathsf{ctxt}^* \leftarrow \mathsf{PEnc}_{\mathsf{pk}_b}(M)$
$b' \leftarrow \mathcal{A}^{\mathsf{PDec}_{\mathsf{sk}_0}, \mathsf{PDec}_{\mathsf{sk}_1}}(\mathsf{ctxt}^*)$
if $b = b'$ then **return** 1 **else return** 0

Figure 1.8: PKE-IK-CCA security for PKE PKE.

Definition 4. *For any PPT adversary $\mathcal{A}$ the advantage function*

$$\mathsf{Adv}^{\mathsf{pke\text{-}ik\text{-}cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda) := \left|\Pr\left[\mathsf{Exp}^{\mathsf{pke\text{-}ik\text{-}cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right|,$$

is negligible in λ, where the experiment $\mathsf{Exp}^{\mathsf{pke\text{-}ik\text{-}cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda)$ is given in Figure 1.8 and PKE is a PKE as above.

We note that for both notions we presented the single-challenge notions. Using a hybrid argument, both can be extended to multi-challenge notions, e.g., see [BBM00].

1.2.6 Digital Signatures

A signature scheme Σ consists of the PPT algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$, which are defined as follows:

$\mathsf{Gen}(1^\lambda)$: On input security parameter λ outputs a signing key sk and a verification key pk with associated message space $\mathcal{M}$.

$\mathsf{Sign}_{\mathsf{sk}}(M)$: On input, a secret key sk and a message $M \in \mathcal{M}$, outputs a signature σ.

$\mathsf{Verify}_{\mathsf{pk}}(M, \sigma)$: On input a public key pk, a message $M \in \mathcal{M}$ and a signature σ, outputs a bit b.

$$\textbf{Exp.}\ \mathsf{Exp}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}}(\lambda)$$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$(M^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}_{\mathsf{sk}}}(\mathsf{pk})$
if $\mathsf{Verify}(\mathsf{pk}, M^*, \sigma^*) = 1$ then **return** 1 **else return** 0

Figure 1.9: The EUF-CMA experiment for a signature scheme Σ.

We assume that a signature scheme satisfies the usual (perfect) correctness notion, i.e. for all security parameters $\lambda \in \mathbb{N}$, for all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$, for all $m \in \mathcal{M}$, we have that

$$\Pr\Big[\mathsf{Verify}_{\mathsf{pk}}(M, \mathsf{Sign}_{\mathsf{sk}}(M)) = 1\Big] = 1.$$

We say a signature Σ is EUF-CMA-secure if and only if any PPT adversary $\mathcal{A}$ has only negligible advantage in the following security experiment. First, $\mathcal{A}$ gets a honestly generated public key and outputs a message M^* and signature σ^*. During the experiment $\mathcal{A}$ has access to an singing oracle $\mathsf{Sign}_{\mathsf{sk}}$ where the adversary can query signatures for arbitrary messages. The experiment outputs 1 if and only if $\mathsf{Verify}_{\mathsf{pk}}(M^*, \sigma^*) = 1$ and M^* was not queried to the signing oracle.

Definition 5. *For any PPT adversary $\mathcal{A}$, we define the advantage in the EUF-CMA experiment $\mathsf{Exp}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}}$ (cf. Figure 1.9) as*

$$\mathsf{Adv}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}}(\lambda) := \Pr\left[\mathsf{Exp}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}}(\lambda) = 1\right].$$

A signature scheme Σ is EUF-CMA-secure, if $\mathsf{Adv}^{\mathsf{euf-cma}}_{\Sigma,\mathcal{A}}(\lambda)$ is a negligible function in λ for all PPT adversaries $\mathcal{A}$.

1.2.7 Diffie-Hellman Assumptions

Subsequently, we recall the strong Diffie-Hellman (SDH) assumption and the oracle Diffie-Hellman assumption (ODH) where the hash function is modeled as a random oracle which is implied by SDH in the ROM [BFGJ17]. Specifically, we consider the mmPRF-ODH assumption where the PRF is instantiated with a random oracle.

Definition 6 (SDH)**.** *The strong Diffie-Hellman assumption holds relative to $\mathcal{G} = (\mathbb{G}, q, g)$ and an oracle $\mathsf{stDH}_{\mathsf{x}}(g^y, g^z)$ that returns 1 if and only if $xy = z$, if for all PPT adversaries $\mathcal{A}$, there is a negligible function ε such that*

$$\Pr\left[\begin{array}{l} x, y \xleftarrow{\$} \mathbb{Z}_q \\ h^* \leftarrow \mathcal{A}^{\mathsf{stDH}_{\mathsf{x}}(\cdot,\cdot)}(g^x, g^y) \end{array} : h^* = g^{xy}\right] \leq \varepsilon(\kappa).$$

Definition 7 (ODH)**.** *The ODH assumption holds relative to* $\mathcal{G} = (\mathbb{G}, q, g)$ *and an oracle* $H \colon \mathbb{G} \to \{0,1\}^\lambda$*, if for all PPT adversaries* $\mathcal{A}$*, there is a negligible function* ε *such that*

$$\left| \Pr \left[\begin{array}{ll} x, y \xleftarrow{\$} \mathbb{Z}_q, b \xleftarrow{\$} \{0,1\} & \\ t^* \leftarrow \mathcal{A}^{H_x}(g^x) & \\ \begin{cases} w \xleftarrow{\$} \{0,1\}^\lambda \ \textit{if } b = 0 \\ w \leftarrow H(g^{xy}, t^*) \ \textit{otherwise} \end{cases} & : b = b^* \\ b^* \leftarrow \mathcal{A}^{H_x, H_y}(g^x, g^y, w) & \end{array} \right] - \frac{1}{2} \right| \leq \varepsilon(\kappa)$$

where $H_x(h,t) = H(h^x, t)$ *and* $H_y(h,t) = H(h^y, t)$ *and the adversary may not query* H_x *on* (g^y, t^*) *and* H_y *on* (g^x, t^*)*, respectively.*

Definition 8 (GDH)**.** *The GDH assumption (see [OP01]) holds relative to* $\mathcal{G} = (\mathbb{G}, q, g)$ *if the following problem cannot be solved by any PPT adversary* $\mathcal{A}$*: Given a triple* (g, g^a, g^b) *find the element* $C = g^{ab}$ *with the help of a Decision Diffie-Hellman Oracle (which answers whether a given quadruple is a Diffie-Hellman quadruple or not).*

1.2.8 Security Proofs: Sequence of Games

In cryptographic proofs, an often used schema is to specify a sequence of games. The first game is the original security game as defined for the property that should be proven. Usually the adversary has to win this game with non-negligible advantage, in order to break the security property. Each subsequent game is then a copy of the previous one with a small change. The final game can then usually be trivially evaluated. In order to complete the proof, one then has to show that all changes between two subsequent games can only be detected with negligible probability by the adversary that plays that game. To be precise, of all the cases in which the adversary wins the first game, the probability that the adversary notices the change to the second game must be negligible. This change from one game to another is called “Game Hop”.

The proof argument then goes as follows: Assume for contradiction that some adversary $\mathcal{A}$ is able to win Game 1 with non-negligible probability. Since with non-negligible probability, $\mathcal{A}$ cannot notice the game hop to Game 2, $\mathcal{A}$ also wins Game 2 with non-negligible probability. This argument continues to some Game N. Usually the adversary cannot win such a Game N for obvious reasons, which yields a contradiction. Hence we conclude that the initial assumption of an adversary winning Game 1 is false.

For a more detailed explanation we refer the reader to [Sho04].

1.3 Preliminaries Tamarin

This section is not intended as a full tutorial on all of Tamarin’s capabilities. For that please refer to available online material such as [Tea]. However it gives a brief overview that should highlight and explain the language constructs that are used in the later chapters of this thesis.

```
theory Example
begin

functions: KE_m1/1, KE_m2/2, KE_kA/2, KE_kB/2
equations: KE_kA(randA, KE_m2(randB, KE_m1(randA))) = KE_kB(
    randB, KE_m1(randA))

// ... (Rules)

end
```

Listing 1.1: Functions and Equations

Listing 1.1 shows the *theory* “Example”. Every file should contain one theory, that is specified with a name as well as the *begin* and *end* commands. The example also shows how functions and equations can be used to model a cryptographic primitive, namely unauthenticated two-move key exchange. *KE_m1* and *KE_m2* are used for the messages of the two parties (i.e. g^x and g^y in the case of DHKE), *KE_kA* and *KE_kB* are used for both sides deriving the session key. Functions are defined with their name and arity. Equations can relate terms. In this case, we define that the resulting session key should be the same for both sides.

```
theory Example
begin

builtins: hashing, asymmetric-encryption, signing, symmetric-
    encryption

// ... (Functions/Equations)

// ... (Rules)

end
```

Listing 1.2: Builtins

Listing 1.2 shows how to activate some predefined functionality of Tamarin. *Builtins* always provide functions and equations for the specific topic, e.g. *signing* contains the functions *pk*, *sign* and *verify* as well as an equation, which is used to model the basic behavior of digital signatures with public and private keys.

```
rule CreateIdentity:
let
    caSig = sign(<~id, pk(~ltk_Sign)>, ltk)
in
```

```
[ Fr(~id), Fr(~ltk_Sign), Fr(~ltk_AEnc), !CA(ltk, pub) ]
--[CreatedParty(~id, ~ltk_Sign, ~ltk_AEnc)]->
[ !Party(~id, ~ltk_Sign, ~ltk_AEnc, caSig),
    Out(~id), Out(pk(~ltk_Sign)), Out(pk(~ltk_AEnc)), Out(caSig
    ) ]
```

Listing 1.3: Rules

Listing 1.3 shows how to create rules in Tamarin. Rules are used to specify the protocol steps. The example defines the rule *CreateIdentity* that adds new users to the experiment. The *let ... in* statements define macros, i.e. all instances of *caSig* are replaced with *sign(<~id, pk(~ltk_Sign)>, ltk)*. In the first square brackets [·] the inputs are specified, together with the initialization of fresh variables (denoted by $Fr(\cdot)$) like *~id*. Fresh variables are used to model randomly drawn values and their names always start with ~. Inputs can also be facts like *!CA(ltk, pub)*. Facts are created by rules and consumed by rules, unless they are marked with *!* in order to define them as persistent facts. Action facts like *CreatedParty* are created whenever the rule is executed and can be used to reason about rule executions in lemmas or restrictions, which are introduced later. In the final [·], the output values are defined. On the one hand, these can be messages sent to the network (by using *Out*). On the other hand these can be facts.

```
rule CA_Init:
[ Fr(~ltk) ]
--[CA_Init()]->
[ !CA(~ltk, pk(~ltk)) ]

restriction CA_Init_Once:
"
    All #i #j . CA_Init() @ #i & CA_Init() @ #j ==> #i = #j
"
```

Listing 1.4: Restrictions and Lemmas

Listing 1.4 defines the rule *CA_Init*, that is used to model one Certificate Authority (CA) that signs the keys of all users. It uses the predefined function *pk* by the *signing* builtin. The example also shows a restriction that says *CA_Init* may only occur once. This is encoded with the first order logic statement "For all timestamps i and j, if CA_Init() happened at timestamp i and CA_Init() happened at timestamp j, then $i = j$". The same syntax can also be used for a *lemma*. In that case, the truth of the statement is evaluated instead of being enforced like with the *restriction*.

```

rule TestPrivacy:
[ !Party(a, k1, k2, caA), !Party(b, l1, l2, caB), Fr(~c), !CA(
   sk, pk) ] --[ Test(), TestPriv(a, b, ~c) ]->
```

```
[ !Party(~c, diff(k1, l1), diff(k2, l2), sign(<~c, diff(k1, l1)
    >, sk)), Out(~c) ]
```

Listing 1.5: Observational Equivalence

Listing 1.5 shows how to use the *diff* operator. This operator causes Tamarin to evaluate observational equivalence, i.e. it checks whether an adversary could notice the difference between two worlds, one in which all diff terms are replaced with their first argument and the other world in which the second argument is used.

1.4 Preliminaries ProVerif

This section is not intended as a full tutorial on all of ProVerif's capabilities. For that please refer to available online material such as [BBS]. However it gives a brief overview that should highlight and explain the language constructs that are used in the later chapters of this thesis.

ProVerif is a tool for automated verification. Users encode their security protocols as well as the desired security properties in ProVerifs specification language. ProVerif then automatically checks whether the desired properties hold. If not, an attack is shown. However, ProVerif can only use the specified functions. If a function that is need for an attack is not defined in the input file, ProVerif might not be able to find an attack on a protocol, even though it has security flaws, as discussed before (see Section 1.1.2).

In this thesis we focus on ProVerif's typed pi calculus (.pv) file format. There are several others available, which however are less useful for our application. An input file consists of

1. initial definitions, such as type definitions, constants and functions,
2. desired security properties, encoded as queries and
3. the actual protocol specification, encoded as processes.

For easier understanding, we will explain the initial definitions and the protocol specification before detailing how to specify security properties.

Initial Definitions

Consider the following example of some initial definitions.

```
type key.
fun senc(bitstring, key): bitstring.
fun sdec(bitstring, key): bitstring.
equation forall m: bitstring, k:key; sdec(senc(m,k),k) = m.
```

Listing 1.6: Defining symmetric encryption

In Listing 1.6 we show one possibility of how to encode symmetric encryption. In line 1 we define a new type, called *key*. Then we define the functions *senc* and *sdec* that take two arguments as input, the first being of the predefined type *bitstring* and the second of type *key*. Both output a *bitstring*. The intended functionality is that users call *senc(m, k)* in order to apply symmetric encryption with key k on message m. Similarly, *sdec* is supposed to be called with some ciphertext and the appropriate key and produce the initial message as output. In line 4 we specify this behaviour in the form of an equation.

Notice that we never specified how *senc* works. In ProVerif, all functions are assumed to reveal absolutely no information about their input. In reality, a symmetric encryption scheme might leak the length of the message m. However this is not the case in ProVerif unless we explicitly model this, e.g. by adding a function *getLength*.
Further notice that all functions are assumed to be publicly known (unless marked as private) and deterministic. Consider the setting in which a client applies asymmetric encryption to some message it sends to the server. If we model the ciphertext as *aenc(m, pubKey)* and the adversary knows m is one of two possibilities m_1, m_2, they may recompute *aenc*(m_i, *pubKey)* for $i \in \{1, 2\}$ and compare it to the received ciphertext. This effectively allows the adversary break the ciphertext indistinguishability in this model. In real-world asymmetric encryption schemes this is impossible since *senc* is required to be a probabilistic function. If we want to model such functions, they should take an additional argument that is used to provide the necessary randomness, i.e. *aenc(m, pubKey, randomness)*.

```
type key.
fun senc(bitstring, key): bitstring.
reduc forall m: bitstring, k:key; sdec(senc(m,k),k) = m.
```

Listing 1.7: Defining symmetric encryption (alternative version)

In Listing 1.7 we show an alternative way to encode symmetric encryption. Instead of modelling *sdec* as proper function, it is now modelled with the *reduc* construct. This is useful when the function symbol, in this case *sdec*, is only used to reduce complex terms to smaller terms. In this variation *sdec* cannot be applied to arbitrary bitstrings like in the previous example. Instead, *sdec(x)* will fail if x is not of the proper form.

```
type name.
const a : name.
```

Listing 1.8: Constants

In Listing 1.8 we show how to create a constant a of type *name*. This might be useful if we want to model a multi-user setting in which a specific user should be treated differently.

Specifying the protocol: Processes

The actual protocol is specified as process. Let us consider the following example.

```
1 fun someFunc(bitstring) : bitstring.
2 channel c.
3
4 process
5     new x : bitstring;
6     let y = someFunc(x) in
7     out(c, y)
```

Listing 1.9: The main process

In Listing 1.9 we first define *someFunc*. Then we use *channel c* (which is equivalent to *const c : channel*) to create a channel constant. Channels represent the network and can be public (default) or private (using *[private]*). Any message sent to a public channel can be intercepted by the adversary, who can also modify messages or send its own messages (i.e. full Man-In-The-Middle capabilities). In line 4 we use the keyword *process* to start the definition of the main process. This is a sequence of operations. First, a new variable x is initialized with a random bitstring. Then the variable y is initialized to the output of *someFunc(x)* using the *let ... in* keywords. Finally y is transmitted over the channel c using *out(c, y)*, effectively revealing it to the adversary.

One may define subprocesses as follows:

```
channel c.

let sender(x: bitstring) =
    new z : bitstring;
    out(c, (x, z)).

let receiver() =
    in(c, msg : bitstring);
    let (x: bitstring, z: bitstring) = msg in
    in(c, (a: bitstring, b: bitstring));
    out(c, (a, b, x, z)).

process
    new x : bitstring;
    sender(x) | receiver()
```

Listing 1.10: Defining process macros

Listing 1.10 illustrates how process macros can be defined using "*let [...]]([...]) = [...].*". ProVerif basically copies the code to all occurrences of the process macro name (with the

variables being renamed if necessary). The process macros *sender* and *receiver* model two different communication parties, where *sender* is called with the parameter x, draws a random z and outputs the tuple *(x, z)*. Tuples are always of type *bitstring*. The process macro *receiver* first receives a message of type *bitstring*. Then it attempts to decode the tuple by using the *let ... in* construct. This can also be used to attempt to match some variable against a term with function calls. We could use *else* to specify behaviour in case the decoding fails. Currently this process macro simply aborts its execution. In line 10 we show a compact formulation that captures the behaviour of line 8 and 9. Finally line 11 outputs a new tuple with four values.
The main process creates a random value x. Then it calls the process macros in parallel using "|".

Specifying security properties

In order to specify security properties that deal with the protocol's execution, e.g. saying that in some protocol, the received messages of one party were sent by another honest party, we need to define events.

```
1  channel c.
2
3  event matchingBits(bitstring). (* Define the event interface *)
4
5  query x : bitstring; event(matchingBits(x)). (* Security
       property *)
6
7  process
8      new x : bitstring;
9      in(c, y : bitstring);
10     if x = y then
11         event matchingBits(x) (* Cause the event *)
```

Listing 1.11: Events and queries

In Listing 1.11, line 3 we define the event *matchingBits* which should also record a value of type *bitstring*. Brackets with stars "(* *)" denote comments. Line 5 states a reachability query in order to define a security property (see detailed explanation of query types below). The process creates a random bitstring x, receives some value y and compares them. In case they are equal, the previously defined event is called.

In this protocol, it should be impossible for the adversary to reach the event *matchingBits*, since the value x is never revealed. This is encoded in the query in line 5. It basically asks "Can the adversary find some bitstring x and somehow interact with the protocol such that the event *matchingBits(x)* is called?" In a regular human-written security proof, one would argue that the probability of guessing x is negligible if x is from a sufficiently large set. ProVerif (with this input file) instead does not even model guessing. It actually tries to somehow deterministically derive or compute the needed value x from the output of the process (which is none in this case) or other public information. Therefore ProVerif will determine this protocol to be fully secure.

Security properties are specified in three ways, one of which is the reachability query that was used in Listing 1.11:

1. **Reachability Query.**
   Using *query var:VARTYPE, var2:VARTYPE2 [, ...]; event(eventName(var, var2, const1))* a reachability query may be specified. This query is written above the protocol specification. It may contain existentially quantified variables (*var:VARTYPE, var2:VARTYPE2 [, ...]*) and constants (*const1*). Such queries are always interpreted as "in a secure protocol, this should not be reachable", i.e. if ProVerif manages to reach the event, it will output "query [...] is false". Furthermore, by adding " *& varX <> varY* " (meaning: and varX is not equal to varY) or similar clauses, it is possible to specify relations between the variables.

2. **Implication Query.**
   Queries can also be specified like this: *query [...]; event(evName1([...])) ==> event(evName2([...]))* . This should be read as whenever *evName1* occurs, there has been an appropriate occurence of *evName2* before. Hence if ProVerif manages to reach *evName1* without triggering *evName2* before, it will output "query [...] is false".

3. **Observation Equivalence.**
   This functionality is not specified using a *query* statement above the protocol specification. Instead *diff* (or equivalently *choice*) is used inside of the protocol specification. By writing *diff[termA, termB]* we tell ProVerif to create two protocols, one with *termA* and one with *termB*. Multiple uses of *diff* do not cause the creation of more protocols, instead the first protocol varation uses all left terms inside of *diff[termA, termB]* statements and the second protocol varation uses all right terms. If at least one *diff* occurs in the protocol specification, ProVerif will attempt to prove observational equivalence. This means that for any adversary and any adaptive interaction with the protocol, both protocol varations behave equivalently. This means that any chosen *if-then* path and any format errors (i.e. a tuple of two values was expected, but only a single value was received) is the same for both variations. The only exception are *if-then* switches inside of terms. Furthermore all terms the adversary receives should be indistinguishable. Consider the following example. The adversary receives $f(a, x)$ (i.e. the output of some function f applied to a and x) in the first variation and $f(b, x)$ in the second variation. We assume that a and b are known to the adversary. Observational equivalence holds if and only if x is not known to the adversary. Otherwise the adversary can recompute

$f(a, x)$ and check whether the result matches the received term, which is only true in one of the variations.

CHAPTER 2

PPAKE Model

2.1 Overview

In order to formally analyze security protocols, it is necessary to first specify the security model, i.e. how do we represent different parties, what capabilities does the adversary have and which security properties should protocols fulfill.

In Section 2.2 we informally discuss the main goals that our model should achieve, i.e. which real-world scenarios it should represent and what kind of attacks should be prevented. In Section 2.3 we describe the model and all of its features in detail, as well as the security definitions. This is then compiled into a short summary in Section 2.4. Finally in Section 2.5 we address questions as to why some design choices were made and why we decided against certain other possibilities.

2.2 Design Goals

In order to explain the design of the model, we first describe what the protocols should achieve. We aim to create an authenticated key exchange (AKE) protocol, that does not leak the identities of the communicating parties to potential adversaries of different strength.

Real World Setting

A real-world use case could be some client contacting a server over some network (e.g. the Internet), in order to establish a common secret key that will then be used for secure communication. Aside from protecting the content of the communication, the client additionally wants to hide who is talking to whom. We assume that the underlying network does not leak the identities, either by nature of the network or through the use of additional tools that are outside the scope of this model (e.g. in case of the

Internet, through the use of VPN services or Tor [DMS04]). Even in this setting, many AKE protocols will still reveal the communicating parties' identities. This is due to the messages of the protocol allowing adversaries to determine the identities.

The focus of our privacy notions is hence to prevent leaking identity related information through the content of the messages. Furthermore our model should contain the standard AKE security notion of key indistinguishability.

Modelling Privacy

We have to somehow formally capture the notion of privacy. Privacy means that an adversary cannot determine who is talking to whom. We model this in the strongest possible way: the adversary only has to deanonymize one communication partner, and only differentiate between two possible parties for that communication partner (instead of determining the correct identity out of a large set of possibilities). Furthermore, these two parties are chosen by the adversary. This is implemented by giving the adversary access to an oracle $\mathsf{Test}(m)$. The adversary will pass two parties, let us denote them by P_i and P_j, to $\mathsf{Test}(m)$ which then creates a new party $P_{i|j}$ which either behaves like P_i or P_j, depending on some secret random bit.

Attacks

Concretely, we want to address the following potential attacks:

1. **Passive eavesdropping.** This is the most basic attack. Adversaries that simply passively listen to the communication should not be able to determine the identities of the parties.

2. **Actively impersonating a party (without any secret keys).** A MITM adversary might intercept all messages that are sent by a party and attempt to answer them directly. Due to the lack of the secret information of the intended party, e.g. some signing key, the adversary is unable to complete the protocol run of many protocols. However, unless the protocol is carefully designed, even such an incomplete protocol run might leak the identity of the first party or its intended peer.

3. **Cross tunnel attack.** The goal of this attack is to deanonymize a party that acts as a responder in the protocol, even though an analogous attack can be executed against the initiator.

   Consider Figure 2.1. The adversary knows that two sessions are active. One between A and the unknown party, which could be B or C. The other session is known to be between A and B. The adversary reroutes the messages, so that B/C (from the first session) is communicating with A (from the second session). In case that B/C is B, the cross tunnel attack still results in an accepted session. If there

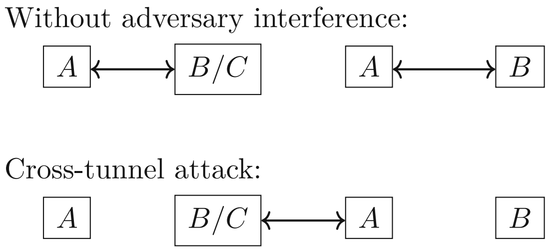

Figure 2.1: Cross tunnel attack: At the top the figure shows how the four protocol instances intend to communicate. At the bottom it shows the effect of the adversaries interference.

is any error, the session is aborted or one party tries to re-initiate the session, then the adversary knows B/C corresponds to C.[1]

4. **Future corruptions.** In this attack scenario, the adversary recorded all communication between two parties, where it does not know the identity of one of the parties. At some point in the future, some parties in the system (potentially all), leak their secret keys. The adversary will now attempt to determine the unknown identity in the previously recorded communication.

Among other reasons, we specifically chose these attacks, since “Actively impersonating a party (without any secret keys)” and “Cross tunnel attacks” are not considered in the models in the literature and break their proposed protocols (e.g. [Zha16], [SSL20], see Section 3.3). We build our model in a modular way which allows us to both evaluate protocols that prevent such attacks as well as protocols that do not.

We aim to prevent other potential attacks that were not listed by formulating the security properties rather general. Furthermore we examine the feasibility of preventing all mentioned attacks if the adversary is able to corrupt parties.

This leads to the following novel security notions that our model incorporates:

- **(Weak) MITM Privacy.** When attempting to break (weak) MITM privacy of a protocol, the adversary is able to act as a MITM in all communication. Aside from the protocol messages themselves it may also use error messages that might be sent by the communicating parties, e.g. because some message was malformed or if a specific session was aborted.[2] (Weakly) MITM private protocols hence prevent the following previously discussed attacks: Passive eavesdropping, actively impersonating a party (without any secret keys), cross tunnel attacks.

[1] In order to prevent this type of attack, our protocols are designed to not behave noticeably different in case of an error, and instead send dummy messages. Clearly, higher level protocols also need to implement a similar behavior in order to fully prevent this attack.

[2] Time is not part of our model. While in a real-world setting the fact that one user takes particularly long to answer might be useful information to an adversary, this is outside the scope of our model.

- **Strong MITM Privacy.** This notion strengthens the adversaries capabilities by allowing the corruption of users, as long as it does not yield a trivial, protocol independent, attack. Strongly MITM private protocols hence also prevent the following previously discussed attacks: Passive eavesdropping, actively impersonating a party (without any secret keys), cross tunnel attacks.
- **Forward Privacy.** Similar to commonly examined forward secrecy, this notion models the setting in which a protocol session happened without interference. Later, an adversary that recorded the information also gains access to some secret keys. The identities of the communicating parties should still remain secret. Forward private protocols hence prevent the following previously discussed attacks: Passive eavesdropping, future corruptions.

Since we model AKE protocols, the standard security property, i.e. *key indistinguishability* is also incorporated into our model. Furthermore we create a specific security property, called *completed-session privacy*, that should capture the notions already present in the literature (i.e. [Zha16] and [SSL20]).

2.3 Model Definition

We build upon the model of [CCG$^+$19] and extend it with additional concepts that allow the evaluation of privacy related notions. The model is parameterized with the number of parties $\mu \in \mathbb{N}$ and their number of sessions $\ell \in \mathbb{N}$. In [CCG$^+$19] these values are used to give concrete security bounds, i.e. breaking a specific protocol takes at least f time, where f is a function depending on μ and ℓ. In this work however we only consider asymptotic security, i.e. if a PPT adversary could break specific protocols. For this reason, μ and ℓ can simply be considered arbitrary integers greater than 1 and polynomially bounded in the security paramenter λ. Hence a PPT adversary is able to iterate over all parties in the system.

2.3.1 Communication Model

We consider μ parties $1, \ldots, \mu$. Each party P_i is represented by a set of oracles, $\{\pi_i^1, \ldots, \pi_i^\ell\}$, where each oracle corresponds to a session, i.e., a single execution of a protocol role, and where $\ell \in \mathbb{N}$ is the maximum number of protocol sessions per party. Each oracle π_i^s is equipped with a randomness tape r_i^s containing random bits, but is otherwise deterministic. Each oracle π_i^s has access to the long-term key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$[3] of party P_i and to the public keys of all other parties, and maintains a list of internal state variables that are described in the following:

- $\mathsf{Pid}_\mathsf{i}^\mathsf{s}$ ("peer id") stores the identity of the intended communication partner. We assume the initator of a protocol to know who she contacts when sending her first

[3]While modeled as a single key pair, in a concrete protocol the private/public keys might be a tuple that contains various private and public keys for signatures and encryption.

message, hence for the initator this value is set at the start of the protocol run. Due to the nature of PPAKE the responder might not immediately know the identity of the initiator, hence for the responder this value is initialized to $\emptyset$ and only set once he receives a message containing the initiator's identity.

- $\Psi_i^s \in \{\emptyset, \mathsf{Accept}, \mathsf{Reject}\}$ indicates whether π_i^s has successfully completed the protocol execution and "accepted" the resulting key.
- k_i^s stores the session key computed by π_i^s
- $\mathsf{role}_i^s \in \{\emptyset, \mathsf{Initiator}, \mathsf{Responder}\}$ indicates π_i^s's role during the protocol execution.

For each oracle π_i^s these variables are initialized to the empty string $\emptyset$. The computed session key is assigned to the variable k_i^s if and only if π_i^s reaches the Accept state, that is we have $k_i^s \neq \emptyset \Leftrightarrow \Psi_i^s = \mathsf{Accept}$. Furthermore the environment maintains two initially empty lists lists $\mathsf{L}_{\mathsf{corr}}$, $\mathsf{L}_{\mathsf{Send}}$ and $\mathsf{L}_{\mathsf{SessKey}}$ of all corrupted parties, sent messages and session keys respectively.

2.3.2 Security Experiment

The security experiment $\mathsf{Exp}_{\mathrm{PPAKE},\mathcal{A}}^{\mathrm{X}}$ is defined as follows.

1. Let μ be the number of parties in the game and ℓ the number of sessions per user. The challenger $\mathcal{C}$ begins by drawing a random bit $b \xleftarrow{\$} \{0,1\}$ and generating key pairs $\{(sk_i, pk_i) | 1 \leq i \leq \mu\}$ as well as oracles $\{\pi_i^s | 1 \leq i \leq \mu, 1 \leq s \leq \ell\}$.
2. $\mathcal{C}$ now runs $\mathcal{A}$, providing all the public keys as input. During its execution, $\mathcal{A}$ may adaptively issue the queries defined below (see Section 2.3.3). While doing so, $\mathcal{A}$ is under some restrictions that are defined later (see Section 2.3.5).
3. The game ends when $\mathcal{A}$ terminates with output b', representing the guess of the secret bit b. If $b' = b$, output 1. Otherwise output 0.

2.3.3 Oracles Available to the Adversary

The adversary $\mathcal{A}$ interacts with the oracles through queries. It is assumed to have full control over the communication network, modeled by a $\mathsf{Send}(i, s, m)$ query which allows it to send arbitrary messages to any oracle. The adversary is also granted a number of additional queries that model the fact that various secrets might get lost or leaked. The queries are described in detail below.

- $\mathsf{Send}(i, s, m)$: This query allows $\mathcal{A}$ to send an arbitrary message m to oracle π_i^s. The oracle will respond according to the protocol specification and its current internal state. To start a new oracle, the message m takes the form:

($\texttt{START}$: role, j): If π_i^s was already initialized before, return $\perp$. Otherwise this initializes π_i^s in the role role, having party P_j as its intended peer. Thus, it sets $\mathsf{Pid}_\mathsf{i}^\mathsf{s} := j$ and $\mathsf{role}_i^s := \mathsf{role}$. If π_i^s is started in the initiator role ($\mathsf{role} = \mathsf{Initiator}$), then it outputs the first message of the protocol.

All $\mathsf{Send}(i, s, m)$ calls are recorded in the list $\mathsf{L}_{\mathsf{Send}}$.

- $\mathsf{RevLTK}(i)$: For $i \leq \mu$, this query returns the long-term private key sk_i of party P_i. After this query, P_i and all its protocol instances π_i^s (for any s) are said to be *corrupted* and P_i is added to $\mathsf{L}_{\mathsf{corr}}$.

- $\mathsf{RegisterLTK}(i, \mathsf{pk}_i)$: For $i > \mu$, this query allows the adversary to register a new party P_i with the public key pk_i. The adversary is not required to know the corresponding private key. After the query the pair (i, pk_i) is distributed to all other parties. Parties registered by $\mathsf{RegisterLTK}(i, \mathsf{pk}_i)$ (and their protocol instances) are corrupted by definition and are added to $\mathsf{L}_{\mathsf{corr}}$.

- $\mathsf{RevSessKey}(i, s)$: This query allows the adversary to learn the session key derived by an oracle. If $\Psi_i^s = \mathsf{Accept}$, return k_i^s. Otherwise return a random key k^* and add (π_i^s, k^*) to $\mathsf{L}_{\mathsf{SessKey}}$. After this query π_i^s is said to be revealed.

  If this query is called for an oracle π_i^s, while there is an entry (π_j^t, k^*) in $\mathsf{L}_{\mathsf{SessKey}}$, so that π_i^s and π_j^t have matching conversations, then k^* is returned.[4]

Additionally, it is given access to a special query $\mathsf{Test}(m)$, which, depending on a secret bit b chosen by the challenger, either returns real or random keys (for key indistinguishability) or an oracle to communicate with one of two specified parties in the sense of a left-or-right oracle for the privacy notions. The goal of the adversary is to guess the bit b. The adversary is only allowed to call $\mathsf{Test}(m)$ once and we distinguish the following two cases:

- Case $m = (\mathsf{TestKeyIndist}, i, s)$: If $\Psi_i^s \neq \mathsf{Accept}$, return $\perp$. Else, return k_b where $k_0 = k_i^s$ and $k_1 \xleftarrow{\$} \mathcal{K}$ is a random key. After this query, oracle π_i^s is said to be tested.

- Case $m = (X, i, j)$, $X \in \{\mathsf{Test\text{-}w\text{-}MITMPriv}, \mathsf{Test\text{-}s\text{-}MITMPriv}, \mathsf{TestForwardPriv}, \mathsf{TestCompletedSessionPriv}\}$: Create a new Party $P_{i|j}$ with identifier $i|j$. This party has all properties of P_i (if $b = 0$) or P_j (if $b = 1$), but no active sessions. The public key of $P_{i|j}$ is not announced to the adversary and the query $\mathsf{RevLTK}(i|j)$ always returns $\perp$. Furthermore create exactly one session $\pi_{i|j}^1$. Return the new handle $i|j$.

[4]Note that the bookkeeping and consistent answers for matched sessions are required to avoid trivial distinguishers in case of cross tunnel attacks (cf. Section 2.2).

2.3.4 Preliminary Definitions

Partnering

We use the following partnering definitions (cf. [CCG+19]). Note that no-match attacks (cf. [LS17]) are prevented by our protocols by including the full transcript in the key derivation.

Definition 9 (Origin-oracle)**.** *An oracle π_j^t is an origin-oracle for an oracle π_i^s if $\Psi_j^t \neq \emptyset$, $\Psi_i^s = \mathsf{Accept}$ and the messages sent by π_j^t equal the messages received by π_i^s, i.e., if* $\mathsf{sent_j^t} = \mathsf{recv_i^s}$.

Definition 10 (Partner oracles)**.** *We say that two oracles π_i^s and π_j^t are partners if (1) each is an origin-oracle for the other; (2) each one's identity is the other one's peer identity, i.e.,* $\mathsf{Pid_i^s} = j$ *and* $\mathsf{Pid_j^t} = i$*; and (3) they do not have the same role, i.e.,* $\mathsf{role}_i^s \neq \mathsf{role}_j^t$.

Oracle Status: Corrupted, Revealed, Fresh

As defined above, a party P_i is called *corrupted* if (a) the adversary used $\mathsf{RevLTK}(i)$ or (b) P_i was created by calling $\mathsf{RegisterLTK}(i, \mathsf{pk}_i)$. An oracle π_i^s is corrupted if its party P_i is corrupted.

An oracle π_i^s is called revealed if $\mathsf{RevSessKey}(i, s)$ was called, as defined above.

Definition 11 (Freshness)**.** *An oracle π_i^s is fresh if*

1. $\mathsf{RevSessKey}(i, s)$ *has not been issued*
2. *no query* $\mathsf{RevSessKey}(j, t)$ *has been issued, where π_j^t is a partner of π_i^s.*
3. $\mathsf{Pid_i^s}$ *was:*
   a) *not corrupted before π_i^s accepted if π_i^s has an origin-oracle, and*
   b) *not corrupted at all if π_i^s has no origin-oracle.*

2.3.5 Adversary Restrictions

During the game, the provided queries may be used any number of times, except for $\mathsf{Test}(m)$, which may be queried only once. Depending on what argument X the $\mathsf{Test}(m)$ oracle was called with, we require the corresponding property below to hold through the entire game.

1. $\mathsf{TestKeyIndist}$: The tested oracle remains fresh (cf. Definition 11).
2. $\mathsf{Test\text{-}w\text{-}MITMPriv}$: No oracle is ever corrupted.

3. Test-s-MITMPriv: P_i and P_j are never corrupted. Furthermore we require that $\mathsf{Pid}^1_{\mathsf{i|j}} = \emptyset$ or $\mathsf{Pid}^1_{\mathsf{i|j}} = k$ for some k, while P_k is never corrupted.

4. TestForwardPriv: The returned oracle $\pi^1_{i|j}$ has a partner oracle π^r_k at the end of the game. Furthermore no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i|j}$.

5. TestCompletedSessionPriv: The returned oracle $\pi^1_{i|j}$'s state is Accept at the end of the game. Let $k = \mathsf{Pid}^1_{\mathsf{i|j}}$. P_k are not corrupted, $\mathsf{RevSessKey}(i|j, 1)$ was never queried and $\mathsf{RevSessKey}(k, r)$ (for any π^r_k that has matching conversations) was never queried.

   Furthermore no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i|j}$.

2.3.6 Security Definitions

The above model can be parameterized by allowing or prohibiting the different types of $\mathsf{Test}(m)$ queries. This leads to the following:

Definition 12. *A key-exchange protocol Γ is called X if for any PPT adversary $\mathcal{A}$ with access to the oracle $\mathsf{Test}(m)$ with queries of the form defined below, the advantage function*

$$\mathsf{Adv}^X_\Gamma(\lambda) := \left| \Pr\left[\mathsf{Exp}^X_{PPAKE,\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2} \right|$$

is negligible in λ, where

- $\mathcal{A}$ *queries* TestKeyIndist*: X = secure.*
- $\mathcal{A}$ *queries* Test-w-MITMPriv*: X = 2-way MITM private.*
- $\mathcal{A}$ *queries* Test-s-MITMPriv*: X = strongly 2-way MITM private.*
- $\mathcal{A}$ *queries* TestForwardPriv*: X = forward private.*
- $\mathcal{A}$ *queries* TestCompletedSessionPriv*: X = completed-session private.*

In the above definition, secure corresponds to having indistinguishable session keys, weak forward secrecy and security against key compromise impersonation (KCI). In order to model explicit entity authentication, we use the following definition.

Definition 13 (Matching Conversation)**.** *Let Π be an N-message two-party protocol in which all messages are sent sequentially.*

- *If a session oracle π^s_i sent the last message of the protocol, then π^t_j is said to have matching conversations to π^s_i if the first $N-1$ messages of π^s_i's transcript agrees with the first $N-1$ messages of π^t_j's transcript.*

- *If a session oracle π_i^s received the last message of the protocol, then π_j^t is said to have matching conversations to π_i^s if all N messages of π_i^s's transcript agrees with π_j^t's transcript.*

We now define implicit authentication through the fact that even a MITM adversary would not be able to derive the session key. This can be done in two moves. Explicit authentication is characterized by the fact that, additionally to providing implicit authentication, the protocol fails if a party does not possess a valid secret key (i.e. an active MITM adversary).

Definition 14 (Explicit entity authentication). *On game* $\mathsf{PPAKE}_{\mathcal{A}}^{\text{2-way-priv}}$ *define* $\mathsf{break}_{\mathsf{EA}}$ *to be the event that there exists an oracle* π_i^s *for which all the following conditions are satisfied.*

1. π_i^s *has accepted, that is,* $\Psi_i^s = \mathsf{Accept}$.
2. $\mathsf{Pid}_\mathsf{i}^\mathsf{s} = j$ *and party* j *is not corrupted.*
3. *There is no oracle* π_j^t *having:*
   a) *matching conversations to* π_i^s *and*
   b) $\mathsf{Pid}_\mathsf{j}^\mathsf{t} = i$ *and*
   c) $\mathsf{role}_j^t \neq \mathsf{role}_i^s$

Definition 15. *A key-exchange protocol* Γ *has explicit authentication, if, for any PPT adversary* $\mathcal{A}$*, the event* $\mathsf{break}_{\mathsf{EA}}$ *(see Definition 14) occurs with at most* $\mathsf{negl}(\lambda)$ *probability.*

2.4 Model Summary

Below we informally summarize all oracles that the adversary can access.

Oracles [Recap]

- $\mathsf{Send}(i, s, m)$: Sending messages.
- $\mathsf{RevLTK}(i)$: Revealing long-term keys (corrupting).
- $\mathsf{RegisterLTK}(i, \mathsf{pk}_i)$: Creating new parties (immediately corrupted).
- $\mathsf{RevSessKey}(i, s)$: Reveal computed session key. Returns a random key if π_i^s has not accepted.
- $\mathsf{Test}(m)$: One-time query to choose security game (i.e. key indistinguishability or some privacy notion). Returns a real-or-random key (key indistinguishability) or the handle for $\pi_{i|j}^1$ (privacy notions).

Below we recall the different security properties. The "goal" is what the adversary must accomplish in order to break the specified security. While attempting to do so, the adversary must not violate the "restrictions". These boxes will be shown again at the beginning of proofs that they are relevant for.

Explicit Authentication [Recap]

Goal: There is some π_i^s s.t.

1. π_i^s has accepted, that is, $\Psi_i^s = \mathsf{Accept}$.
2. $\mathsf{Pid}_\mathsf{i}^\mathsf{s} = j$ and party j is not corrupted.
3. There is no oracle π_j^t having:
   a) matching conversations to π_i^s and
   b) $\mathsf{Pid}_\mathsf{j}^\mathsf{t} = i$ and
   c) $\mathsf{role}_j^t \neq \mathsf{role}_i^s$

Key Indistinguishability [Recap]

Goal: Return bit b, indicating wether the given key was a real or a random key.

Restrictions: The tested oracle remains fresh (cf. Definition 11).

weak MITM Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: No oracle is ever corrupted.

strong MITM Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: P_i and P_j are never corrupted. Furthermore we require that $\mathsf{Pid}_{\mathsf{i|j}}^1 = \emptyset$ or $\mathsf{Pid}_{\mathsf{i|j}}^1 = k$ for some k, while P_k is never corrupted.

Forward Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: The returned oracle $\pi^1_{i|j}$ has a partner oracle π^r_k at the end of the game. Furthermore no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i|j}$.

Completed Session Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: The returned oracle $\pi^1_{i|j}$'s state is Accept at the end of the game. Let $k = \mathsf{Pid}^1_{\mathsf{i|j}}$. P_k are not corrupted. Also $\mathsf{RevSessKey}(i|j, 1)$ was never queried and $\mathsf{RevSessKey}(k, r)$ (for any π^r_k that has matching conversations) was never queried.

2.5 Model Discussion

In Section 2.5.1 we discuss some relations between the presented security notions. Some of these relations show that the different properties are actually describing different classes of protocols, i.e. for a pair of properties there are some protocols that achieve one property but not the other. Furthermore the section shows some implications between some properties, which are useful since they mean that for concrete protocols, not all properties have to be explicitly shown. In the following sections we discuss several design choices regarding the model, including why some potential extensions where not implemented.

2.5.1 Relations Between the Privacy Properties

Subsequently, we investigate the relations between the different privacy notions.

Lemma 2. *Strong 2-way MITM privacy is strictly stronger than (weak) 2-way MITM privacy.*

Proof. This immediately follows from the tighter restrictions put on the attacker in the (weak) 2-way MITM privacy test. Furthermore, there are protocols that are (weak) 2-way MITM anonymous but not strongly 2-way MITM anonymous (see e.g. Figure 3.2 in Section 3.2). □

Lemma 3. *The 2-way MITM privacy notions are independent of forward privacy.*

Proof. Note that the privacy notions do not allow corruptions of the test oracle and forward privacy does not allow the attacker modify any sent messages (i.e. does not allow the attack to act as an active MITM). For example Π^2_{PKE} (see Section 3.2) is strongly 2-way MITM private and hence also (weakly) 2-way MITM private, but it is not forward private as the identities are only encrypted using long term keys. On the other hand a protocol that runs the classic Diffie-Hellman key exchange followed by transmitting their identities symmetrically encrypted would reach forward privacy, but no 2-way MITM privacy, as any MITM adversary could simply run the protocol, pretending to be the intended peer. □

Completed-session privacy is implied by the other privacy notions: if a protocol is strong MITM private or has explicit authentication and is forward private, then it also provides completed session-privacy. The following lemma shows the implication starting from strong MITM privacy.

Lemma 4. *Let Γ be a PPAKE protocol. If Γ is strong MITM private, then it is completed-session private.*

Proof. Strong 2-way MITM privacy test puts less restrictions on the attacker. □

Finally, the following Theorem covers completed-session privacy from explicit authentication forward privacy.

Theorem 1. *Let Γ be a PPAKE protocol. If Γ has explicit authentication and is forward private, then it is completed-session private.*

Proof. Assume for contradiction that some Γ has explicit authentication and is forward private, but is not completed-session private. This means a PPT-adversary $\mathcal{A}$ is able to call $\mathsf{TestCompletedSessionPriv}$ and not violate the imposed restrictions, while also correctly guessing the challenge bit b with non-negligible probability. Since Γ is forward private, the adversary violates a necessary restriction for calling $\mathsf{TestForwardPriv}$ while correctly guessing the challenge bit b. (Note that otherwise the exact same adversary $\mathcal{A}$ breaks forward privacy by simply using the argument $\mathsf{TestForwardPriv}$ instead).

Since both notions do not allow to call the oracle $\mathsf{RevSState}(i|j, 1)$, this part of the restriction for using $\mathsf{TestForwardPriv}$ is fulfilled. It follows that after $\mathcal{A}$ is done, $\pi^1_{i|j}$ does not have a partner oracle with non-negligible probability (which is the only other way to violate the restriction for calling $\mathsf{TestForwardPriv}$). As per requirement of winning with $\mathsf{TestCompletedSessionPriv}$, there is the oracle $\pi^1_{i|j}$ which has accepted and party P_k, where $k = \mathsf{Pid}^1_{\mathsf{i|j}}$, is not corrupted. Due to Γ providing explicit authentication, there is an oracle π^r_k s.t. π^r_k has matching conversations to $\pi^1_{i|j}$, $\mathsf{Pid}^\mathsf{r}_\mathsf{k} = i|j$ and $\mathsf{role}^r_k \neq \mathsf{role}^1_{i|j}$ (see Def. 14 detailing explicit entity authentication). Then $\mathcal{A}$ could simply not drop the last message (if it did before) thereby making $\pi^1_{i|j}$ and π^r_k have matching conversations to each other. This also makes $\pi^1_{i|j}$ and π^r_k be partnered to each other, without making it less likely for $\mathcal{A}$ to correctly guess the challenge bit b. Hence $\mathcal{A}$ is able to break forward privacy, which is a contradiction. $\square$

2.5.2 Partnering

In this work we define partnering with regards to origin-oracles. There is an alternative variation based on original keys by Li and Schäge [LS17], which addresses a potential model-theoretical attack they discovered called no-match attack. This attack is prevented by our protocols by using the full transcript for key derivation, a generic countermeasure that was also described by Li and Schäge [LS17].

2.5.3 One-way privacy

Note that the second case of the query $\mathsf{Test}(m)$ produces $\pi^1_{i|j}$ which can be used as an initiator or a responder. This means that we model two-way privacy. In case of one-way privacy, i.e. the privacy only holds either for the initiator or the responder (depending on the protocol), we need to restrict the adversary s.t. the first message sent to $\pi^1_{i|j}$ via $\mathsf{Send}(i|j, 1, m)$ must be a `START` command. Analogously, we can model scenarios where we only consider privacy of the responder involved in a session.

2.5.4 Revocation

In our model, corruptions are immediately publicly known. While this is an idealization, defending against secret corruptions is infeasible, since an adversary could perfectly impersonate the corrupted user.

In a real-world implementation, corruptions could be handled by using revocation. There is previous work that formally models revocation for AKE protocols [BCF+13], but we decided against this possibility (as typically done in AKE). The reason is that we want to avoid the additional complexity of the model and we also do not want to restrict the implementations to one specific revocation mechanism. We note however that for any revocation mechanism, the revocation status of a communication partner can only be checked after they revealed their identity. For this reason, we model strong MITM privacy so that the adversary can corrupt users as long as it does not openly identify itself as that user.

2.5.5 Completed Session Privacy

As mentioned in the design goals (Section 2.2), completed session privacy is meant to capture the privacy definitions seen in the literature ([SSL20] and [Zha16]). We do not give a proof that the representation is accurate and leave that to the readers inspection. It should be noted that we made the additional restriction "no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i|j}$". This is a necessary addition since due to the nature of our model there are otherwise trivial attacks against a large class of protocols:

First of all the adversary makes the test oracle complete its session without interfering and hence fulfills the experiment's requirements. It then corrupts both of the test oracle's possible identities. Finally it instructs a new oracle to initiate the protocol with the test oracle being the intended recipient, but answers all messages itself using the information obtained with the corruptions. If the imitator at any point uses the intended recipient's public key, e.g. for PKE, then the adversary learns the test oracle's identity.

This problem does not exist in the model of Zhao [Zha16] since it does not inform the initiator about the responders identity. It also does not exist in the model of Schäge et al. [SSL20], since they let each initiator determine the identity of the test oracle (if configured correspondingly), instead of having the identity of the test oracle fixed throughout the entire experiment. We note that while [SSL20] always model two identities per party, in our model every party only has a single identity.[5]

2.5.6 Weak MITM Privacy

As mentioned in the design goals (Section 2.2), weak MITM privacy should prevent cross tunnel attacks. Indeed, if such an attack is possible against some protocol then that protocol cannot provide MITM privacy. The reason is that our model allows the adversary to create the exact situation needed for cross tunnel attacks. They simply use $\mathsf{Test}(m)$ to create $\pi^1_{i|j}$ and use $\mathsf{Send}()$ to instruct some other oracle to initiate the protocol run with intended peer P_i, but redirect the messages to $\pi^1_{i|j}$.

Furthermore weak MITM privacy should prevent attackers from simply running the protocol, obtaining the identity of either the initiator or the responder, and then aborting the protocol run. Indeed, if such an attack is possible against a protocol, an adversary in our model can utilize this method to break MITM privacy. Note that this is the reason why the protocols that are discussed by Zhao [Zha16] and Schäge et al. [SSL20] do not provide MITM privacy (see discussion in Section 3.3).

2.5.7 Strong MITM Privacy

As mentioned in the design goals (Section 2.2), strong MITM privacy represents security against the same attack scenarios as weak MITM privacy while allowing as much additional corruptions as possible. However it is still necessary to prohibit the corruption of $\pi^1_{i|j}$'s peer. Clearly if an adversary has corrupted some party P_k, they can simply execute a normal protocol run while perfectly imitating P_k, since we do not model revocations. As explained in Section 2.5.1 we do allow the adversary to corrupt some party P_k and use its information against $\pi^1_{i|j}$, as long as $\mathsf{Pid}^1_{\mathsf{i|j}}$ is not set.

[5] Clearly, one could however group parties to generate virtual parties with more identities in our model though.

Furthermore the adversary cannot be allowed to corrupt P_i or P_j. Otherwise it might instruct some unrelated oracle π_k^r to initiate the protocol run with intended peer $P_{i|j}$, but intercept the messages and answer by running the protocol with the secret key of P_i (P_j, respectively). If the protocol run does not succeed, the adversary knows that $P_{i|j}$ corresponds to P_j (P_i, respectively).

2.5.8 Forward Privacy

As mentioned in the design goals (Section 2.2), forward privacy should capture the setting in which the adversary records communication between two parties and later learns the secret keys of some (or all) parties in the system. We model this by fully allowing corruptions. In order to simulate a previously recorded completed session, we require $\pi_{i|j}^1$ to have a partner oracle (which implies that there was no interference during the communication).

Furthermore we require that no oracle besides the partner oracle may be instructed to start communicating with $\pi_{i|j}^1$. This is needed to prevent the same trivial attack as discussed in Section 2.5.5.

CHAPTER 3

PPAKE Protocols

In this chapter we present several protocols and prove their security in the previously introduced model. In Section 3.1 we present the main result of this work, a 4-move PPAKE that achieves all privacy definitions of the model. Since the full privacy might not be needed for certain applications, in Section 3.2 we examine protocols which only fulfill some privacy properties, but in turn provide a reduced round complexity. In Section 3.3 we examine recent protocols of the literature in our model and discuss why they do not achieve our strongest privacy notions. Finally this chapter is concluded in Section 3.4.

3.1 Protocol Π_{Gen}

In this section we present the main contribution of this work, namely the protocol Π_{Gen}. It is a 4-move protocol that fulfils all security and privacy properties of the model.

3.1.1 Protocol Definition

In Figure 3.1 we present the protocol Π_{Gen}.

Certificates. In our protocol we write Cert_A to indicate a tuple consisting of A's name, A's public key and the CA's signature on that information.[1] As is usually done, the cryptographic guarantees are not based on keeping Cert_X secret, where X is some honest user. Instead, it should be hard for the adversary to produce a new, valid Cert_X on their own (in particular for parties that the adversary creates on their own).

Theorem 2. *The protocol Π_{Gen} in Figure 3.1 provides explicit authentication, and is secure, strongly MITM private and forward private, if KE Γ is unauthenticated and*

[1] We note that in small-scale systems, in which all users keep a table of all authentic public keys, Cert_A can also be realized as simply being the name A.

secure, the PKE PKE *is PKE-IND-CCA- and PKE-IK-CCA-secure, symmetric encryption scheme* Ω *is LH-SE-IND-CCA-secure, and the signature scheme* Σ *is EUF-CMA-secure.*

The proofs of the individual properties are given in Section 3.1.3 and the subsequent sections.

Alice		**Bob**
(check if B is revoked)		
	$\xrightarrow{m_1 = \Gamma(0)}$	
	$\xleftarrow{m_2 = \Gamma(1, m_1)}$	
$x \xleftarrow{\$} \{0,1\}^\lambda$		
$k' \leftarrow H(\Gamma.key, x, \mathsf{ctxt})$		
	$m_3 = (c_0 = E_{\Gamma.key}(\mathsf{PEnc}_B(x)), c_1 = E_{k'}(\mathsf{Cert}_A, \mathsf{Sign}_A(A\|\|B\|\|c_0\|\|\mathsf{ctxt})))$ $\longrightarrow$	
		Attempt:
		$x \leftarrow \mathsf{PDec}_B(D_{\Gamma.key}(c_0))$
		$k' \leftarrow H(\Gamma.key, x, \mathsf{ctxt})$
		$(\mathsf{Cert}, \mathsf{Sign}) \leftarrow D_{k'}(c_1)$
		Validate Cert and Sign
		if any attempted step failed or Cert is revoked: send random m_4. otherwise
	$\xleftarrow{m_4 = H(x, \mathsf{ctxt}_2)}$	
(Internally verify m_4)		
$k \leftarrow H(\Gamma.key, x, \mathsf{ctxt}_3)$		$k \leftarrow H(\Gamma.key, x, \mathsf{ctxt}_3)$

Figure 3.1: Protocol Π_{Gen}, using an unauthenticated KE Γ, PKE $\mathsf{PKE} = (\mathsf{PEnc}, \mathsf{PDec})$, symmetric encryption $\Omega = (E, D)$, signature scheme $\Sigma = (\mathsf{Sign}, \mathsf{Verify})$, Cert_A as discussed in Section 3.1.1, $\mathsf{ctxt} = m_1||m_2$, $\mathsf{ctxt}_2 = A||B||m_1||m_2||m_3$, and $\mathsf{ctxt}_3 = A||B||m_1||m_2||m_3||m_4$.

3.1.2 Protocol Discussion

This section gives an informal description of Π_{Gen} and the purpose of each of its components.

Achieving Forward Secrecy and Privacy
Related components: m_1, m_2, *Unauthenticated Key Exchange* Γ

m_1 and m_2 represent the execution of the unauthenticated key exchange protocol Γ (e.g. Diffie-Hellman Key Exchange). This part of the protocol on its own is susceptible to MITM attacks, which is why it is designed to depend solely on ephemeral randomness (and no long-term, i.e. identity related, information). The computed key $\Gamma.key$ is used to provide forward secrecy and privacy, since even if all secret keys are leaked later, $\Gamma.key$ cannot be recovered.

Defense Against MITM Attackers (1/2)
Related components: *Nonce* x*, Public Key Encryption* $\mathsf{PEnc}, \mathsf{PDec}$

The random nonce x can only be known to the initiator and the intended responder, as it was encrypted using the responders public key. Its main purpose is to fend off MITM attackers.

Defense Against MITM Attackers (2/2)
Related components: k', m_3 *(*c_0*), Symmetric Encryption* $\Omega = (E, D)$*, Key Privacy*

Since $\Gamma.key$ can be produced by a MITM adversary, we introduce k' which also depends on the aforementioned x. Since x is not known to the responder before receiving m_3, m_3 is split into two parts, c_0 and c_1.

c_0 contains x, which is encrypted twice. The public key encryption one the one hand is used to authenticate B (as mentioned in the previous section) and on the other hand prevents MITM adversaries from obtaining x. Since PKE is key private (PKE-IK-CCA-secure), even a MITM adversary that computed $\Gamma.key$ cannot learn the intended responders identity from c_0. The public key ciphertext is again encrypted using symmetric encryption with $\Gamma.key$. This is needed to provide forward privacy. (Recall: forward privacy considers an adversary which obtains all secret keys.) The reason is that key privacy does not hold if the secret keys are leaked.

Authenticating A and Guaranteeing Equal Transcripts
Related components: m_3 *(*c_1*)*, Cert, *Signature Scheme* $\Sigma = (\mathsf{Sign}, \mathsf{Verify})$

c_1 contains the certificate Cert_A, which is some data that is sufficient to (a) convince B that there is an honest user A in the system and (b) allow B to obtain that honest user's public key (see discussion in Section 3.1.1). Note that Cert_A of an honest user A could have been sent by an adversary since it is easy to obtain. For this reason we additionally use signatures.

Indeed, c_1 contains a signature on all information to this point. This is used to authenticate the initiator and guarantee that both sides agree on the transcript so far.

Authenticating B and Guaranteeing Equal Transcripts
Related components: m_4, k

The final message m_4 is used to authenticate the responder. Since x is needed to produce m_4, only the initiator's intended peer can create m_4, as discussed before. The hash value m_4 also depends on the context ctxt_2, which ensures that both parties agree on the transcript. This also prevents the adversary from recording m_4 and using it to impersonate B in a different communication, as the adversary is then forced to use the same m_2 (without knowing the randomness that is needed to derive $\Gamma.key$). Furthermore its structure (simply a hash value) allows the protocol to easily fake m_4, which can only be detected by the party that sent m_3 (since x is needed to correctly compute m_4).

Finally the resulting key k is derived from all information that was used in the protocol run.

3.1.3 Proof: Explicit Authentication

Lemma 5. Π_{Gen} *in Figure 3.1 provides explicit authentication if* Γ *is a secure unauthenticated two-move key exchange protocol,* PKE *is a PKE-IND-CCA secure public-key encryption scheme,* Ω *is a LH-SE-IND-CCA secure symmetric encryption scheme and* Σ *is an EUF-CMA secure signature scheme.*

Oracles [Recap]

- $\mathsf{Send}(i, s, m)$: Sending messages.
- $\mathsf{RevLTK}(i)$: Revealing long-term keys (corrupting).
- $\mathsf{RegisterLTK}(i, \mathsf{pk}_i)$: Creating new parties (immediately corrupted).
- $\mathsf{RevSessKey}(i, s)$: Reveal computed session key. Returns a random key if π_i^s has not accepted.
- $\mathsf{Test}(m)$: One-time query to choose security game (i.e. key indistinguishability or some privacy notion). Returns a real-or-random key (key indistinguishability) or the handle for $\pi_{i|j}^1$ (privacy notions).

Explicit Authentication [Recap]

Goal: There is some π_i^s s.t.

1. π_i^s has accepted, that is, $\Psi_i^s = \mathsf{Accept}$.
2. $\mathsf{Pid_i^s} = j$ and party j is not corrupted.
3. There is no oracle π_j^t having:
   a) matching conversations to π_i^s and
   b) $\mathsf{Pid_j^t} = i$ and
   c) $\mathsf{role}_j^t \neq \mathsf{role}_i^s$

Proof. Assume for contradiction that $\mathcal{A}$ breaks explicit authentication, i.e. for some π_i^s, that has accepted and its peer $j = \mathsf{Pid_i^s}$ is not corrupted, there is no π_j^t that has matching conversations. We view the two cases of π_i^s's role separately.

Case 1. $\mathsf{role}_i^s = \mathsf{Initiator}$. It follows that π_i^s has received a valid m_4. Except with $\mathsf{negl}(\lambda)$ probability, this means that $H(x, \mathsf{ctxt}_2)$ was queried. Note that m_3 is the only available source to reproduce x.

- Game 0: The original game.
- Game 1: Guess i, s, j. Abort if wrong.
- Game 2: Let x be the value that π_i^s computes for sending m_3. (This is determined before the game using π_i^s's randomness tape.) Pick x^* randomly so that $||x^*|| = ||x||$. Modify π_i^s to use $c^* = \mathsf{PEnc}_j(x^*)$ instead of $\mathsf{PEnc}_j(x)$ in its message. Modify all instances π_j^t of P_j to not actually decrypt c^* but instead treat x as the result of the decryption. (Hence in this game all oracles act as if x was still used everywhere, except that m_3 and hence the ctxts have changed. Note that m_3 is now independent of x.)

Notice that in Game 2, x is only ever used as input to the RO. Hence the adversary can only guess x. Also the adversary or an oracle must produce $m_4 = H(x, \mathsf{ctxt}_2)$. If an oracle used the correct ctxt_2 this means it has matching conversations to π_i^s and agrees on the identities and roles, which contradicts the initial assumption. Therefore the adversary must have guessed x or m_4 and the probability of winning Game 2 is $\mathsf{negl}(\lambda)$.

Indistinguishability of game hops.

- Game 0 $\to$ Game 1: This guessing leads to a polynomial loss of winning probability.
- Game 1 $\to$ Game 2: If $\mathcal{A}$ notices this change, we can break PKE-IND-CCA-security of the PKE. For this, modify Game 2 as follows: Let pk, PDec be the public key

and oracle provided by the PKE-IND-CCA-challenger. Set $\mathsf{pk}_j = \mathsf{pk}$. All messages m_3 sent to instances of P_j can be decrypted using the challenger's oracle PDec. Replace c^* sent by π_i^s with the ciphertext c obtained from the PKE-IND-CCA-challenger for the messages $a_0 = x, a_1 = x^*$. Notice that $\mathsf{PDec}(c)$ is never queried as due to the definition of Game 2. Now if the challengers bit $b = 0$, this game is behaving identical to Game 1. If $b = 1$, then this game behaves identical to Game 2. Therefore, our constructed adversary against PKE-IND-CCA-security outputs 1 if $\mathcal{A}$ notices the Game Hop from 1 to 2. Otherwise output a random bit.

Case 2. $\mathsf{role}_i^s = \mathsf{Responder}$. Then π_i^s received a valid m_3, which contains $\sigma = \mathsf{Sign}_j(j||i||c_0||\mathsf{ctxt})$.

Case 2a. Assume some π_j^t computed σ at any point. It follows π_j^t has matching m_1, m_2, c_0. Since m_1, m_2 are matching, $\Gamma.key$ is also matching. Since c_0 is matching, x and hence k' is also matching. The only way that π_j^t does not have matching conversations is if π_j^t produced a different c_1.

We show that this is only possible if $\mathcal{A}$ can break LH-SE-IND-CCA security of E_k or the EUF-CMA security of Sign. Let E, D be the oracles provided by some LH-SE-IND-CCA-challenger.

- Game 0: The original game.
- Game 1: Guess i, s, j, t. Abort if wrong.
- Game 2: π_i^s and π_j^t use a random k'.
- Game 3: π_j^t, instead of outputting c_1, outputs c_1^* which is received from the LH-SE-IND-CCA-challenger for $(a_0 = (\mathsf{Cert}_j, \mathsf{Sign}_j(A||B||c_0||\mathsf{ctxt})), a_1)$ where a_1 is a random message. π_i^s uses D for decryption and treats a_1 as verifying. (a_0 will verify just like any other pair $(\mathsf{Cert}_j, \mathsf{Sign})$ where Sign is a valid signature by P_j.)

Indistinguishability of game hops.

- Game 0 $\rightarrow$ Game 1: This guessing leads to a polynomial loss of winning probability.
- Game 1 $\rightarrow$ Game 2: Notice both have matching m_1, m_2 i.e. honestly generated $\Gamma(0)$, $\Gamma(1)$. Due to the security of Γ we have an indistinguishable-from-random $\Gamma.key$ and consequently an indistinguishable-from-random k'.
- Game 2 $\rightarrow$ Game 3: In case m_3^* is the encrypted a_0, this change is unobservable. Hence if $\mathcal{A}$ can detect this change, we again break LH-SE-IND-CCA-security. Our constructed adversary against LH-SE-IND-CCA-security outputs $b' = 1$ if $\mathcal{A}$ detects the change and a random bit b' otherwise.

Since π_j^t does not have matching conversations as per assumption, D is never queried for m_3^*. Hence embedding the LH-SE-IND-CCA-challenger's oracles as done above is valid.

Per assumption, the adversary is able to win Game 0. As just shown the adversary therefore also has non-negligible probability to win Game 3. We now discuss how we can utilize this fact to construct an adversary against the LH-SE-IND-CCA security of Ω or the EUF-CMA security of Σ.

- Note that the final game is identical to Game 2 from $\mathcal{A}$'s view, if the LH-SE-IND-CCA-challenger's bit b_C was 0. It follows that $\mathcal{A}$ has non-negligible advantage if $b_C = 0$ (since $\mathcal{A}$ was able to win Game 2).
- Hence $\mathcal{A}$ must have non-negligible advantage as well if $b_C = 1$ (otherwise simply construct an adversary against LH-SE-IND-CCA-security that outputs 0 if $\mathcal{A}$ wins or a random bit otherwise).
- View the case that $b_C = 1$. In order to win, $\mathcal{A}$ needs to produce some c_1^*, $c_1^* \neq c_1$ (where c_1 was produced by π_j^t) and c_1^* is decrypted to a_1 or the pair $(\mathsf{Cert}_j, \Sigma^*)$ for some valid Σ^* of P_j. We distinguish the two cases.
  - **Case 1.** $\mathcal{A}$ produces some c_1 that is decrypted by π_i^s to a_1 with non-negligible probability. This allows us to construct an adversary against the LH-SE-IND-CCA security of the symmetric encryption scheme since a_1 was randomly chosen and can hence only be recovered by the adversary if $b_C = 1$.
  - **Case 2.** $\mathcal{A}$ produces some c_1 that is decrypted by π_i^s to $(\mathsf{Cert}_j, \Sigma^*)$ (where Σ^* is a valid signature) with non-negligible probability, we will show that this means that $\mathcal{A}$ was able to break EUF-CMA.

    First of all, start an EUF-CMA challenger to receive pk^* and gain access to the oracle Sign. Since we now attack EUF-CMA security, the LH-SE-IND-CCA challenger is considered part of our game and can be modified.
    * Game 3.0: Our current game, including the LH-SE-IND-CCA challenger.
    * Game 3.1: The LH-SE-IND-CCA challenger always uses $b_C = 1$.
    * Game 3.2: Instead of querying the LH-SE-IND-CCA challenger for (a_0, a_1) as defined before, query it for (a^*, a_1), where a^* is a random message.
    * Game 3.3: Replace the pk of P_j with pk^* and all protocol instances of party P_j use the provided oracle Sign instead of computing signatures themselves.

    Indistinguishable Game Hops:
    * Game 3.0 $\rightarrow$ 3.1: If this is noticeable to $\mathcal{A}$, this yields a trivial distinguisher against the LH-SE-IND-CCA security.
    * Game 3.1 $\rightarrow$ 3.2: This cannot be detected by $\mathcal{A}$, since a_0 is never used by the LH-SE-IND-CCA challenger anyways.

* Game 3.2 → 3.3: Since corruptions of P_j are not allowed, $\mathcal{A}$ cannot notice this change.

Constructing an adversary against EUF-CMA If $\mathcal{A}$ wins Game 3.3, it has to provide a valid signature (as discussed before). The message of this signature was never queried using the Sign oracle, since π_j^t will not have called this query as per game design and other oracles will never arrive at the same ctxt (recall Lemma 1). Hence the signature that $\mathcal{A}$ provided can be used to win the EUF-CMA game.

Case 2b. If no π_j^t produced $\sigma_j(\mathsf{ctxt})$: Break EUF-CMA as illustrated below. Let pk, Sign be given by a EUF-CMA challenger.

- Game 0: The original game.
- Game 1: Guess i, s, j, abort if wrong.
- Game 2: Set $\mathsf{pk}_j \leftarrow \mathsf{pk}$, implicitly setting sk_j to the sk by the EUF-CMA challenger. Any signing operations done by instances of P_j are done by calling Sign.

Our constructed adversary against EUF-CMA outputs the received $\sigma_j(\mathsf{ctxt})$.

Indistinguishability of game hops.

- Game 0 → Game 1: This guessing leads to a polynomial loss of winning probability.
- Game 1 → Game 2: This change is unobservable, since P_j must not be corrupted.

It follows that $\mathcal{A}$ wins Game 2 with non-negligible probability, which also causes our adversary against EUF-CMA to win with non-negligible probability, which is a contradiction. □

3.1.4 Proof: Strong MITM-Privacy

Recall the following theorem:

Theorem 2. *The protocol Π_{Gen} in Figure 3.1 provides explicit authentication, and is secure, strongly MITM private and forward private, if KE Γ is unauthenticated and secure, the PKE PKE is PKE-IND-CCA- and PKE-IK-CCA-secure, symmetric encryption scheme Ω is LH-SE-IND-CCA-secure, and the signature scheme Σ is EUF-CMA-secure.*

strong MITM Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: P_i and P_j are never corrupted. Furthermore we require that $\mathsf{Pid}^1_{\mathsf{i|j}} = \emptyset$ or $\mathsf{Pid}^1_{\mathsf{i|j}} = k$ for some k, while P_k is never corrupted.

Lemma 6. *Π_{Gen} is strongly MITM-private, if KE Γ is unauthenticated and secure, the PKE PKE is PKE-IND-CCA- and PKE-IK-CCA-secure, symmetric encryption scheme Ω is LH-SE-IND-CCA-secure, and the signature scheme Σ is EUF-CMA-secure.*

Proof.

- **Case 1.** $\pi^1_{i|j}$ is $\mathsf{Initiator}$. Therefore $p := \mathsf{Pid}^1_{\mathsf{i|j}}$ is immediately set, and P_p must not be corrupted. Clearly, m_1 and m_2 are independent of the test bit b (i.e. independent of $\mathsf{pk}_{i|j}, \mathsf{sk}_{i|j}$).
  - Game 0: The original game.
  - Game 1: Guess i, j, p. Abort if $\mathsf{Test}(m)$ does not return $i|j$ or $p \neq \mathsf{Pid}^1_{\mathsf{i|j}}$ at the end of the game.
  - Game 2: Replace part of the message m_3 by $\pi^1_{i|j}$ as follows: Instead of sending $c_0 = E_{\Gamma.key}(\mathsf{PEnc}_p(x))$ send $c_0^* = E_{\Gamma.key}(\mathsf{PEnc}_p(z))$ where z is a random number of equal length to x. Program all other oracles to treat c_0^* as c_0 (i.e. the decryption is x).
  - Game 3: Pick k^* randomly. Program all oracles to use k^* instead of computing their original k', if k' should be computed using $k' \leftarrow H(\Gamma.key, x, \mathsf{ctxt})$ where x is the value computed by $\pi^1_{i|j}$ (determined before the game using $\pi^1_{i|j}$'s randomness tape) and $\Gamma.key, \mathsf{ctxt}$ are arbitrary values.
  - Game 4: Replace part of the message m_3 by $\pi^1_{i|j}$ as follows: Instead of sending $c_1 = E_{k^*}(u)$ where $u = (\mathsf{Cert}_A, \mathsf{Sign}_A(A||B||c_0||\mathsf{ctxt}))$ send $c_1^* = E_{k^*}(w)$ for some random w. (Note that E is length hiding.)

  Indistinguishability of game hops.

  - Game 0 $\rightarrow$ Game 1: This guessing leads to a polynomial loss of winning probability.
  - Game 1 $\rightarrow$ Game 2: The indistinguishability of this game hop follows from the PKE-IND-CCA security of PKE, see the proof for Lemma 5.
  - Game 2 $\rightarrow$ Game 3: Since in Game 2, x is only used as input to the RO, the adversary cannot obtain x and hence not check whether the RO actually produced this output. It follows that this change is only detectable with $\mathsf{negl}(\lambda)$ probability as well. (Compare with proof for explicit authentication.)

- Game 3 → Game 4: The indistinguishability of this game hop follows from the PKE-IND-CCA security of the symmetric encryption Ω (see the proof for Π_{Gen} having explicit authentication).

Notice that in Game 4, $\mathsf{pk}_{i|j}, \mathsf{sk}_{i|j}$ were not used at all. Hence in Game 4 all oracles behave independent of b. It follows the probability of $\mathcal{A}$ winning Game 4 is $\frac{1}{2}$.

- **Case 2.** $\pi^1_{i|j}$ is $\mathsf{Responder}$. m_1 and m_2 do not depend on $\mathsf{pk}_{i|j}, \mathsf{sk}_{i|j}$. Below we argue why m_3 does not reveal the key that was used for encryption. m_4 only depends on the key in the sense that it is only valid if m_3 can be decrypted.

  Since PKE is PKE-IK-CCA, we can replace $\mathsf{pk}_{i|j}$ with a random key. To show this, consider the game hops below. Note that a valid or invalid m_4, i.e. $\pi^1_{i|j}$ being able to decrypt m_1 or not, does not give any information about the secret bit b anymore if $\mathsf{pk}_{i|j}$ is replaced with a random key.

  - Game 0: The original game.
  - Game 1: Whenever any oracle sends m_3 with intended recipient $\pi^1_{i|j}$, it saves the message it produced in a secret table together with the data that was encrypted. $\pi^1_{i|j}$, instead of decrypting incoming messages, will look up the content in the secret table. If the message is not in the table, it will attempt to decrypt the message normally.
  - Game 2: $\pi^1_{i|j}$ treats the incoming message m_3 as malformed if there is no matching entry in the secret table. (Note that in this game, under no circumstances $\pi^1_{i|j}$ actually decrypts any messages.)
  - Game 3: Instead of using the public key of i or j (depending on the secret bit b), the party $P_{i|j}$'s public key is set to a randomly generated public key pk'.

  Indistinguishability of game hops.

  - Game 0 → Game 1: This is only a conceptual change.
  - Game 1 → Game 2: This change can only be detected, if $\pi^1_{i|j}$ receives a valid m_3, where m_3 is not the (exact) output of some other oracle. $\pi^1_{i|j}$ would then wrongfully respond with a randomly generated m_4 (since it treats m_3 as malformed) whereas in Game 1 it would respond with a valid m_4.However in these cases, in which $\pi^1_{i|j}$ produces a valid response in Game 1, $\pi^1_{i|j}$ is set to the accept state. Since Π_{Gen} has explicit authentication, it follows that m_3 authenticates a corrupted user (since no oracle has matching conversations to $\pi^1_{i|j}$), except with $\mathsf{negl}(\lambda)$ probability. This means that only in a setting in which the adversary would not have won Game 1 (except with $\mathsf{negl}(\lambda)$ probabilty), it can detect the change to Game 2. Hence we only lose a $\mathsf{negl}(\lambda)$ amount of winning probability in this scenario.
  - Game 2 → Game 3: If this change is detectable with non-negligible probability, then the adversary can break the PKE-IK-CCA security. To show this, use a

PKE-IK-CCA challenger $\mathcal{C}$ and modify our Game 3 so that if $\mathcal{C}$ has chosen secret bit $b_C = 0$ we have Game 2 and if $b_C = 1$ we have Game 3. To do so, we first have to guess i and j that will be used by the adversary in $\mathsf{Test}(m)$ (resulting in polynomial loss of winning probability).

Before the game starts, obtain pk_0 and pk_1 from C. Set $\mathsf{pk}_i = \mathsf{pk}_0$ or $\mathsf{pk}_j = \mathsf{pk}_0$ (depending on the secret bit b), so that $\mathsf{pk}_{i|j} = \mathsf{pk}_0$. Furthermore, set pk' (defined in Game 3) to pk_1. Now any oracle that starts communications with $\pi^1_{i|j}$ does not encrypt its first message on its own, but rather queries the encryption oracle provided by $\mathcal{C}$. If $b_C = 0$ this encrypted message exactly resembles Game 2 and if $b_C = 1$ it exactly resembles Game 3. The behaviour of $\pi^1_{i|j}$ does not need to be changed as it never actually decrypts the incoming messages.

□

3.1.5 Proof: Forward-Privacy

Recall the following theorem:

Theorem 2. *The protocol Π_{Gen} in Figure 3.1 provides explicit authentication, and is secure, strongly MITM private and forward private, if KE Γ is unauthenticated and secure, the PKE* PKE *is PKE-IND-CCA- and PKE-IK-CCA-secure, symmetric encryption scheme Ω is LH-SE-IND-CCA-secure, and the signature scheme Σ is EUF-CMA-secure.*

Forward Privacy [Recap]
Goal: Return bit b, indicating wether the new party $P_{i\|j}$ is equivalent to P_i or P_j.
Restrictions: The returned oracle $\pi^1_{i\|j}$ has a partner oracle π^r_k at the end of the game. Furthermore no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i\|j}$.

Lemma 7. *Π_{Gen} is forward private if KE Γ is unauthenticated and secure, the PKE* PKE *is PKE-IND-CCA- and PKE-IK-CCA-secure, symmetric encryption scheme Ω is LH-SE-IND-CCA-secure, and the signature scheme Σ is EUF-CMA-secure.*

Proof. Consider the following game hops, that end in a game in which the transcript does not depend on b.

- Game 0: The original game.
- Game 1: Guess i, j, k, r. Abort if $\pi^1_{i|j}$ is not partnered to π^r_k at the end.

- Game 2: $\pi^1_{i|j}$ and π^r_k use a random r instead of the computed $\Gamma.key$.
- Game 3: $\pi^1_{i|j}$ and π^r_k use a random k^* instead of k'.
- Game 4: Instead of m_3, the initiator sends $m_3^* = (E_r(z), E_{k^*}(v))$, where z, v are random values. (Note that the PKE ciphertext, the certificate and the signature are removed from the transcript.) The receiver treats m_3^* as if it was the normally computed m_3.

Indistinguishability of game hops.

- Game 0 → 1: This leads to a polynomial loss of winning probability.
- Game 1 → 2: This change cannot be detected with noticeable probability due to the security of Γ. To show this, simply embed the messages produced by the challenger for eavesdropper-security of Γ in the first message of $\pi^1_{i|j}$ and π^r_k. If the attacker is then able to distinguish the computed key of Γ from a random one, it is able to win the $\mathsf{Exp}^{\mathsf{eav}}_{\Gamma,\mathcal{A}}(\lambda)$ game.
- Game 2 → 3: This change cannot be detected, since the input to the RO (specifically r) is hidden.
- Game 3 → 4: This change cannot be detected due to the length-hiding PKE-IND-CCA security of (E, D). To show this, we show that c_0 being replaced is undetectable (c_1 can be treated analogously). Ask some LH-SE-IND-CCA-challenger $\mathcal{C}$ for the normal input for encrypting c_0 as M_0 and $M_1 = z$, receiving ctxt^*. Set $c_0 = \mathsf{ctxt}^*$. If the challengers bit $b_{\mathcal{C}} = 0$ this looks like Game 3 to the adversary. Hence if $\mathcal{A}$ can detect the modification of this game, it can break LH-SE-IND-CCA-security.

□

3.1.6 Proof: Key Indistinguishability

Recall the following theorem:

Theorem 2. *The protocol Π_{Gen} in Figure 3.1 provides explicit authentication, and is secure, strongly MITM private and forward private, if KE Γ is unauthenticated and secure, the PKE* PKE *is PKE-IND-CCA- and PKE-IK-CCA-secure, symmetric encryption scheme Ω is LH-SE-IND-CCA-secure, and the signature scheme Σ is EUF-CMA-secure.*

Key Indistinguishability [Recap]
Goal: Return bit b, indicating wether the given key was a real or a random key.
Restrictions: The tested oracle remains fresh (cf. Definition 11).

Lemma 8. Π_{Gen} *is secure if KE* Γ *is unauthenticated and secure, the PKE* PKE *is PKE-IND-CCA- and PKE-IK-CCA-secure, symmetric encryption scheme* Ω *is LH-SE-IND-CCA-secure, and the signature scheme* Σ *is EUF-CMA-secure.*

Proof. Let π_i^s be the tested oracle. Let $j = \mathsf{Pid}_i^s$. π_i^s must conform to freshness (Definition 11). **Case (a)** Clause 3a was fulfilled, i.e. there is a partner oracle π_j^t. It follows that π_j^t has matching conversations to π_i^s. **Case (b)** Clause 3b was fulfilled, which means j must not be corrupted. Together with the fact that π_i^s accepted, from Π_{Gen} providing explicit authentication follows that there is some π_j^t that has matching conversations to π_i^s. It follows that in any case there is some π_j^t that has matching conversations to π_i^s.

To distinguish the session key k from random, $\mathcal{A}$ needs to query $H(\Gamma.key, x, \mathsf{ctxt}_3)$. We show that $\Gamma.key$ cannot be produced by the adversary.

- Game 0: The original game.
- Game 1: Guess i, s, j, t, abort if guessed wrong.
- Game 2: Before the game, query the challenger for the security of Γ and recieve a transcript (m_1^*, m_2^*) and a key k^*. π_i^s and π_j^t send m_1^*, m_2^* instead of newly computed m_1, m_2 and use k^* instead of $\Gamma.key$.
- Game 3: π_i^s and π_j^t use a random value u instead of k^*.

Indistinguishable Game Hops:

- Game 0 $\rightarrow$ 1: This results in a polynomial loss of winning probability.
- Game 1 $\rightarrow$ 2: If this game hop can be detected, it means that k^* is not the session key that corresponds to the transcript, and hence $\mathcal{A}$ can break Γ's security.
- Game 2 $\rightarrow$ 3: Analogous argument as for Game Hop 1 $\rightarrow$ 2.

In Game 3, $\mathcal{A}$ cannot deduce u since it is only used as input to the RO. Hence $\mathcal{A}$ has $\mathsf{negl}(\lambda)$ chance to win Game 3. □

3.2 Protocols with Reduced Privacy and Round Complexity

In this section we present two protocols that are designed for weaker security guarantees and can hence be formulated with less moves. Furthermore we discuss the whether a one-move PPAKE could be feasible.

3.2.1 Using a shared secret: Π_{ss}

In this section we present a 3-move protocol that utilizes a shared secret to prevent adversaries from eavesdropping. In terms of our model, the shared secret s is part of the secret keys and can hence be compromised by corrupting any party. We construct the protocol so that leaking s does not endanger the usual key indistinguishability.

Alice: $\mathsf{Cert}_A = (A = g^a, \ldots)$ $\mathsf{sk}_A = (a, s)$		**Bob**: $\mathsf{Cert}_B = (B = g^b, \ldots)$ $\mathsf{sk}_B = (b, s)$
$x \xleftarrow{\$} \mathbb{Z}_p$	$\xrightarrow{m_1 = g^x}$	$y \xleftarrow{\$} \mathbb{Z}_p$
		$k' \leftarrow H(g^{xy}, g^x, g^y, s)$
		$c_2 \leftarrow E_{k'}(\mathsf{Cert}_B)$
$k' \leftarrow H(g^{xy}, g^x, g^y, s)$	$\xleftarrow{m_2 = (g^y, c_2, U)}$	$U \leftarrow H(g^{xb}, g^{xy}, g^x, g^y, c_2, s)$
verify B, Cert_B, U		
$c_3 \leftarrow E_{k'}(\mathsf{Cert}_A)$		
$V \leftarrow H(g^{ya}, g^{xy}, g^x, g^y, m_2, s)$	$\xrightarrow{m_3 = (c_3, V)}$	verify Cert_A, V
$k \leftarrow H(g^{xy}, g^{xb}, g^{ya}, s, (m_i)_{i=1}^3)$		$k \leftarrow H(g^{xy}, g^{xb}, g^{ya}, s, (m_i)_{i=1}^3)$

Figure 3.2: Protocol Π_{ss} with shared secret s, using symmetric encryption $\Omega = (E, D)$.

Note the protocol in Figure 3.2 should behave indistinguishable to real executions (from an eavesdropper's view) even if some verification (indicated using boxes) fails, i.e., instead of the real m_3 the encryption of a random message as well as a random hash is returned.

Theorem 3. *If the Oracle Diffie-Hellman (ODH) assumption holds and symmetric encryption scheme Ω is LH-SE-IND-CCA-secure, then Π_{ss} in Figure 3.2 provides explicit entity authentication, is secure, weakly 2-way MITM private and forward private.*

We prove this theorem in two parts as Lemmas 9 and 10.

For the proof of Lemma 9, we require the following definition:

Definition 16 ($\mathsf{LookupOrRandom}(\exists X_1, \ldots, X_k \colon H(V_1, \ldots, V_m), \phi)$)**.** *Let $X_1, \ldots, X_k$ denote variables. Let $V_1, \ldots, V_m$ denote expressions, containing the variables $X_1, \ldots, X_k$. Let ϕ be a logical formula, containing the variables $X_1, \ldots, X_k$. The proof shortcut* $\mathsf{LookupOrRandom}(\exists X_1, \ldots, X_k \colon H(V_1, \ldots, V_m)\phi)$ *is evaluated as follows.*

1. *If there are some $X_1, \ldots, X_k$ s.t. ϕ is true and the RO was queried before for $H(V_1, \ldots, V_m)$, then return this result $H(V_1, \ldots, V_m)$.*
2. *Else, draw Z randomly. Program the RO s.t. when it receives a query of the form $H(V_1, \ldots, V_m)$ for any $X_1, \ldots, X_k$ s.t. ϕ is true, answer Z. Return Z.*

Lemma 9. *If the ODH assumptions holds, then* Π_{ss} *in Figure 3.2 provides explicit authentication.*

Proof. Assume $\mathcal{A}$ breaks explicit authentication.

- **Case 1.** π_i^s is initiator, let m_1 be its first message. Let $j = \mathsf{Pid}_\mathsf{i}^\mathsf{s}$ and $b = \mathsf{sk}_j$, $B = \mathsf{pk}_j$. The proof below is based on the following fact. Since π_i^s accepted, we know it received $m_2 = (Y, c_2, U)$ s.t. $D_{k'}(c_2) = B$ and $H(B^x, g^{xy}, g^x, g^y, c_2, s) = U$. We show that g^{xb} is hard to construct for $\mathcal{A}$. Let (X^*, B^*) be an arbitrary ODH-challenge.

  – Game 0: The original game.

  – Game 1: Before the start of the game, replace pk_j with B^* (thereby also modifying Cert_j). After some π_j^t receives X as the first message, let $U^* = \mathsf{LookupOrRandom}(\exists T\colon H(T, X^y, X, g^y, c_2, s), \mathsf{stDH_y}(X, T) = 1))$. $m_2 = (g^y, E_{k'}(\mathsf{Cert}_j), U^*)$, where g^y and k' is computed normally. Record (U^*, m_1, m_2, k'). π_j^t returns m_2.
  At the end of a protocol run, π_j^t determines its session key k as follows: Let $a = \mathsf{Pid}_\mathsf{j}^\mathsf{t}$, $A = \mathsf{pk}_a$. $k = \mathsf{LookupOrRandom}(\exists T\colon H(X^y, T, A^y, s), \mathsf{stDH_y}(X, T) = 1)$.

  – Game 2: Replace m_1 from π_i^s with X^*. π_i^s, after receiving $m_2 = (Y, c_2, U)$, validate U by checking if (a) (U, m_1, m_2, k') for some k' is in the secret table or (b) the RO produced U for the call $H(T, Z, X^*, Y, c_2, s)$ for any T, Z such that $\mathsf{stDH_x}(B^*, T) = \mathsf{stDH_x}(Y, Z) = 1$ (since B^* is pk_j). In case (b) set $k' = H(Z, X^*, Y, s)$ and $V^* = \mathsf{LookupOrRandom}(\exists Z_2\colon H(Y^a, Z_2, X^*, Y, m_2, s), \mathsf{stDH_x}(Y, Z_2) = 1)$. π_i^s outputs $m_3 = (E_{k'}(\mathsf{pk}_i), V^*)$.
  At the end of a protocol run, π_i^s determines its session key k as follows: Let there be a second secret table. If the second secret table contains an entry for (m_1, m_2, m_3, k) for any k, take that k. Otherwise: Let $a = \mathsf{sk}_i$. $k = \mathsf{LookupOrRandom}(\exists R, O\colon H(R, O, Y^a, s), \mathsf{stDH_x}(Y, R) = \mathsf{stDH_x}(B^*, O) = 1)$. Record (m_1, m_2, m_3).[2]

  Indistinguishability of game hops.

  – Game 0 → Game 1: The change of pk_j cannot be detected from the initial public key distribution since B^* is drawn under the same distribution. The change of U to U^* cannot be detected, as the RO behaves accordingly. Since $\mathsf{Pid}_\mathsf{i}^\mathsf{s}$ must not be corrupted, $\mathsf{RevLTK}(j)$ cannot be called.

  – Game 1 → Game 2: Replacing the first message cannot be detected, since X^* is drawn at the same probability distribution. The validation is indeed indistinguishable from a normal protocol run. k' either corresponds to the value of the other modified oracle or can be computed correctly.

[2] In case both our modified oracles talk, they need to decide on the same random value, hence the second secret table and the check for (m_1, m_2, m_3, k).

Consequences. Notice in Game 2, π_i^s only accepts if it receives U s.t. either (a) (U, m_1, m_2) was put into the secret table by some π_j^t or (b) the RO was called for $H(T, Z, X^*, Y, c_2, s)$ where $\mathsf{stDH_x}(B^*, T) = 1$. Since (a) implies that π_j^t has a matching conversation to π_i^s, it contradicts the initial assumption. On the other hand if (b) holds, we can construct an adversary against the ODH-assumption, by running Game 2 and outputting T.

- **Case 2.** π_i^s is responder, let $m_1 = X, m_3 = (c_3, V)$ be the messages it receives and $m_2 = (g^y, c_2, U)$ be the sent message. Let $j = \mathsf{Pid_i^s}$ and $a = \mathsf{sk}_j, A = \mathsf{pk}_j$. Since π_i^s accepted, we know that $V = H(A^y, X^y, X, g^y, m_2, s)$. This time $A^y = g^{ay}$ is difficult to produce for the adversary.

  Consider analogous game hops to Case 1, again the long-term key of π_j^t (in this case A) and the random group element of π_i^s (in this case g^y) are replaced with the ODH-challenge. The proof that these game hops are indistinguishable to the adversary is analogous. Again, either π_j^t has matching conversations to π_i^s or we can construct an adversary against the ODH-assumption.

□

Lemma 10. *If (E_k, D_k) is a LH-SE-IND-CCA secure symmetric encryption scheme and the DDH assumptions holds, then Π_{ss} in Fig. 3.2 is secure, (weakly) 2-way MITM anonymous and forward private.*

Proof. Assume some PPT adversary $\mathcal{A}$ wins $\mathsf{PPAKE}_{\mathcal{A}}^{\text{2-way-priv}}$ with non-negligible probability. Without loss of generality, assume that only one type of $\mathsf{Test}(m)$ query is issued. (If not, we can construct an adversary $\mathcal{A}_X$ for $X \in \{\mathsf{TestForwardPriv}, \mathsf{Test\text{-}w\text{-}MITMPriv}, \mathsf{TestKeyIndist}\}$ that abort if a $\mathsf{Test}(m)$ query without i is called. At least one of them has non-negligible probability to win $\mathsf{PPAKE}_{\mathcal{A}}^{\text{2-way-priv}}$.) We view the different types of $\mathsf{Test}(m)$ queries separately.

1. Assume $\mathcal{A}$ used $\mathsf{TestKeyIndist}$. Let π_i^s be the tested oracle. In the cases that $\mathcal{A}$ wins, π_i^s conforms to freshness (see Definition 11). If clause 3a is satisfied, we know that there is a partner oracle π_j^t and in particular that π_i^s and π_j^t have matching conversations. If clause 3b is satisfied, we know that $j := \mathsf{Pid_i^s}$ is not corrupted. Also from the $\mathsf{Test}(m)$ query succeeding we know that π_i^s has accepted. Since our protocol provides explicit authentication (see Theorem 9), it follows that there is an oracle π_j^t having matching conversations to π_i^s. Therefore in both cases we have such an oracle π_j^t that has matching conversations to π_i^s.

   - Game 0: The original game.
   - Game 1: Guess s, i for the oracle that $\mathsf{Test}(m)$ will be called with. If guessed wrong: abort.

- Game 2: Guess j, t and abort if π_j^t does not have matching conversations to π_i^s at the end.
- Game 3: Let X, Y, Z be the challenge of a DDH-Challenger. Instead of randomly drawing x, y and thereby calculating g^x, g^y, π_i^s and π_j^t transmit X, Y (notice that due to the matching conversations, these exact values also reach the respective other oracle). Whenever g^{xy} should be used, i.e. the key derivation, instead use Z.

Indistinguishability of game hops.

- Game 0 $\rightarrow$ Game 1: This guessing leads to a polynomial loss of winning probability.
- Game 1 $\rightarrow$ Game 2: This guessing leads to a polynomial loss of winning probability.
- Game 2 $\rightarrow$ Game 3: Since $\mathcal{A}$ is not allowed to call $\mathsf{RevSessKey}(i, s)$ for π_i^s and π_j^t, it is not able to detect the embedding of X and Y, which are also drawn uniformly at random. In order to distinguish the keys k and k' from random or to distinguish them from the correctly calculated ones in Game 2, it needs to call (1) $H(Z, \ldots)$ or (2) $H(g^{xy}, \ldots)$, where x and y are the secret exponents used for X and Y. If $Z \neq g^{xy}$, it was drawn at random and hence cannot be guessed. We conclude $\mathcal{A}$ needed to call $H(g^{xy}, \ldots)$. Under the assumption that $\mathcal{A}$ is able to distinguish k from random or distinguish games 2 and 3, we hence guess which of the $poly(n)$ many queries issued to H by $\mathcal{A}$ contained g^{xy}. Thus, if Z is equal to this value g^{xy}, output 0 to the DDH-Challenger, otherwise output 1.

2. Assume $\mathcal{A}$ used $\mathsf{Test\text{-}w\text{-}MITMPriv}$ and can now interact with $\pi_{i|j}^1$. In order to win, $\mathcal{A}$ must not corrupt any oracles. We show that $\mathcal{A}$ must be able to break LH-SE-IND-CCA-security of $\Omega = (E, D)$.
Case 1. $\pi_{i|j}^1$ did not send its last message (i.e. m_2 or m_3, depending on the role of $\pi_{i|j}^1$). If follows $\mathcal{A}$ can only base its decision on (a) m_1 and (b) the fact that $\pi_{i|j}^1$ might have rejected the previous message (i.e. m_1 or m_2, depending on the role of $\pi_{i|j}^1$). Since (a) does not reveal information (m_1 is random and independent on the test bit b), only (b) is possible. Since m_1 is only rejected if it is malformed, we only need to view the case that m_2 was rejected, i.e. $\pi_{i|j}^1$ is $\mathsf{Initiator}$. Since m_1, m_2 and the corresponding validations are independent on the ID of the initiator, m_2's acceptance/rejection cannot yield any information to the attacker.
Case 2. $\pi_{i|j}^1$ did send its last message. In case $\pi_{i|j}^1$ is initiator, this means $\pi_{i|j}^1$ has accepted. Since Π_{ss} has explicit authentication, this means that there is some partnered oracle π_k^r. Refer to the proof of the case "$\mathcal{A}$ used $\mathsf{TestForwardPriv}$" below. Otherwise, $\pi_{i|j}^1$ is responder and m_2 its only output. Since a fitting U cannot be produced by the adversary due to not knowing s, the adversary cannot modify m_2 in a valid way. Oracles that receive m_2 behave independent of the test bit b (and

hence the $\mathsf{pk}_{i|j}$) except that they might set their state to Reject, which is invisible to the adversary. Therefore only m_2 itself might leak information. We show that this is not the case by implicitly replacing k' with a random k^* (only detectable by solving CDH, formal proof below) and the ciphertext c_2, which depends on the bit b, with a specific ciphertext c_2^* that is independent of b. We show that this indistinguishable to the adversary, unless it is able to break CPA-security of $Omega = (E, D)$.

- Game 0: The original game.
- Game 1: Choose k^* randomly. Replace k' used by $\pi^1_{i|j}$ with k^*. If any oracle has sent the m_1, which was received by $\pi^1_{i|j}$ and receives the modified m_2 by $\pi^1_{i|j}$, it also replaces its k' with k^*.
- Game 2: Choose $m \xleftarrow{\$} \{\mathsf{Cert}_i, \mathsf{Cert}_j\}$ and replace c_2' with $c_2^* = E_{k^*}(m)$.

In Game 3 all challenge bit b related ID information of $\pi^1_{i|j}$ was removed, hence the adversary only has probability 0.5 of winning the game.
Indistinguishability of game hops.

- Game 0 $\rightarrow$ Game 1: Since k' is the output of a random oracle, it is indistinguishable to the attacker that does not know s, since corruptions are not allowed.
- Game 1 $\rightarrow$ Game 2: Due to LH-SE-IND-CCA security of E, this change is not noticeable. To show this, since the used key k^* is random and only used once, we can embed the challenge produced by a LH-SE-IND-CCA-challenger for $(m_0 = \mathsf{Cert}_i, m_1 = \mathsf{Cert}_j)$ in c_2.

3. Assume $\mathcal{A}$ used TestForwardPriv. Hence $\pi^1_{i|j}$ is partnered to some π^r_k, i.e. $\mathcal{A}$ was passive during the communication.

   - Game 0: Original Game.
   - Game 1: Guess i, j, k, r. If wrong, abort.
   - Game 2: Sample k^* randomly. Produce the same transcript, but instead of sending $E_{k'}(cert_X)$ where $X \in \{i, j, k\}$, send $E_{k^*}(cert_X)$. Also replace U and V with random values.
   - Game 3: Replace the ciphertext sent by $\pi^1_{i|j}$ with the result of querying $\mathrm{Test}(cert_i, cert_j)$ at the CPA challenger.

   Since g^x, g^y, U, V, k^* are random, and the ciphertext sent by $\pi^1_{i|j}$ is identical for $b = 0$ and $b = 1$, the adversary has probability 0.5 of winning Game 3.

   Indistinguishability of game hops.

   - Game 0 $\rightarrow$ Game 1: This leads to a polynomial loss.
   - Game 1 $\rightarrow$ Game 2: If detectable, then $\mathcal{A}$ can solve CDH (g^{xy}).

Alice		**Bob**
$x \leftarrow \Gamma(0)$		
$\sigma_A \leftarrow \mathsf{Sign}_A(x, \mathsf{Cert}_B)$	$\xrightarrow{m_1 = \mathsf{PEnc}_B(x, \mathsf{Cert}_A, \sigma_A)}$	decrypt m_1 and verify σ_A
		$y \leftarrow \Gamma(1, x)$
decrypt m_2 and verify σ_B	$\xleftarrow{m_2 = \mathsf{PEnc}_A(y, \sigma_B)}$	$\sigma_B \leftarrow \mathsf{Sign}_B(x, y)$
$k \leftarrow H(\Gamma.key, m_1, m_2)$		$k \leftarrow H(\Gamma.key, m_1, m_2)$

Figure 3.3: Protocol Π^2_{PKE} using a PKE PKE, an unauthenticated KE Γ, and a signature scheme Σ. where Certs contain Σ and PKE public keys.

- Game 2 $\rightarrow$ Game 3: Clearly, $\mathcal{A}$ behaves identical in Game 1 and Game 2 if b chosen by our game is equal to b' chosen by the CPA-challenger, since k'. If $\mathcal{A}$ can detect the game hop with non-negligible probability, then $b \neq b'$ with non-negligible probability. Hence send b to the challenger if $\mathcal{A}$ did not detect the game hop and $1 - b$ otherwise.

On strong MITM privacy. Recall the definition of strong MITM privacy.

strong MITM Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: P_i and P_j are never corrupted. Furthermore we require that $\mathsf{Pid}^1_{\mathsf{i|j}} = \emptyset$ or $\mathsf{Pid}^1_{\mathsf{i|j}} = k$ for some k, while P_k is never corrupted.

This protocol clearly is not strongly MITM private, as corrupting any party P_r will enable the adversary to generate m_1 as per protocol definition, send it to $P_{i|j}$ and then decrypt the received answer m_2. The adversary can hence deanonymize the recipient in the strong MITM privacy experiment. Note that the restriction of strong MITM privacy only prevents the adversary from sending a m_3 that uses P_r's identity, but sending m_3 at all is not required for this attack.

3.2.2 Two move protocol: Π^2_{PKE}

In this section we present a protocol that, contrary to the previously shown protocols, is not designed to provide forward privacy. This allows us to reduce the necessary moves to two. At the same time it even provides strong MITM privacy.

Theorem 4. *If KE Γ is secure, the PKE PKE is length-hiding, PKE-IND-CCA- and PKE-IK-CCA-secure, and the signature scheme Σ is EUF-CMA-secure, then Π^2_{PKE} provides* *explicit entity authentication, is secure, strongly MITM private and completed-session private.*

The theorem is split into Lemmas 11 to 13 and we follow the same strategy as for the proofs regarding Π_{Gen}, hence we may skip some details.

Lemma 11. *If* PKE *is a length-hiding and PKE-IND-CCA secure PKE, and* Σ *is a EUF-CMA secure signature scheme, then* Π^2_{PKE} *in Figure 3.3 provides explicit authentication.*

Proof. Assume for contradiction that $\mathcal{A}$ breaks explicit authentication, i.e., for some π_i^s, that has accepted and its peer $j = \mathsf{Pid}_i^s$ is not corrupted, there is no π_j^t that has matching conversations. We view the two cases of π_i^s's role separately.

Case 1. $\mathsf{role}_i^s = \mathsf{Initiator}$. It follows that π_i^s has received a valid m_2, which contains σ_B. This means there are two cases.

The first case is that no oracle computed σ_B, which means $\mathcal{A}$ breaks the $EUF-CMA$ security of Σ (following a similar argument as the proof of Lemma 5).

The second case is that the adversary used a σ_B that was produced by some π_j^t. We now show that this yields a contradiction.

- Game 0: The original game.
- Game 1: Guess i, s, j. Abort if wrong.
- Game 2: Let x be the value that π_i^s computes for its first message. (This is determined before the game using π_i^s's randomness tape.) Pick a random x^* of equal length. Modify π_i^s to actually send $m_1^* = \mathsf{PEnc}_j(x^*, \mathsf{Cert}_A, \sigma_A)$. Modify all instances of P_j to treat the first argument x^* as x when receiving m_1^*. (Hence in this game all oracles act as if x was still used everywhere, except that m_1, which is now independent of x, has changed.)

  Note that if the adversary wins Game 2, they must have taken σ_B from a message m_2, which was produced by some π_j^t after receiving m_1 (since otherwise there is no way to make π_j^t create a signature that contains x).
- Game 3: When π_j^t would send a signature of x, y (where x is the value that was read from the randomness tape in Game 2, not the transmitted value in m_1 of π_i^s), it now instead sends a random value U of equal length to the actual signature. Program all oracles to now treat U as equivalent to the original signature.

Indistinguishability of game hops.

- Game 0 $\rightarrow$ Game 1: This guessing leads to a polynomial loss of winning probability.

- Game 1 → Game 2: Follows from PKE-IND-CCA security of PKE similar to the proof of Lemma 5.
- Game 2 → Game 3: Follows from PKE-IND-CCA security of PKE similar to the proof of Lemma 5.

Note that in the final game, there is no trace of σ_B in the transcript. Hence if the adversary produces this value, this can be used to attack the $EUF-CMA$ security of Σ like in the first case. On the other hand, if $\mathcal{A}$ sends U to π_i^s, this means that the adversary was able to break PKE-IND-CCA security of PKE (the argument is similar to the proof of Lemma 5).

Case 2. $\mathsf{role}_i^s = \mathsf{Responder}$. Then π_i^s received a valid m_1, which contains $\sigma_j(\mathsf{ctxt})$.

Case 2a. If some π_j^t produced $\sigma_j(\mathsf{ctxt})$, similar to Case 2a. of Lemma 5 we can build an adversary against PKE-IND-CCA security of PKE or EUF-CMA security of Σ.

- Game 0: The original game.
- Game 1: Guess i, s, j, t. Abort if wrong.
- Game 2: π_j^t, instead of outputting m_1, outputs m_1^* which is received from the PKE-IND-CCA-challenger for $a_0 = (x, \mathsf{Cert}_j, \sigma_j(\mathsf{ctxt}))$ and a_1 being a random string (note that PKE is length-hiding). π_i^s uses the decryption oracle for decryption and treats both a_0 and a_1 as verifying.

Since π_j^t does not have matching conversations as per assumption, the decryption oracle is never queried for m_1^*. If in the final game, $\mathcal{A}$ wins, our constructed adversary against PKE-IND-CCA-security outputs b' according to the result of the decryption, i.e. a_0 or a_1. Otherwise it outputs a random bit b'.

Indistinguishability of game hops.

- Game 0 → Game 1: Guess this values incurs a polynomial loss.
- Game 1 → Game 2: In case m_1^* is the encrypted a_0, this change is unobservable. Hence if $\mathcal{A}$ can detect this change, we again break PKE-IND-CCA-security.

Case 2b. If no π_j^t produced $\sigma_j(\mathsf{ctxt})$, we construct and adversary against EUF-CMA of Σ in the same way as in Case 2b of Lemma 5, which we do not repeat here. □

Lemma 12. *If KE Γ is unauthenticated and secure, and PKE PKE is length-hiding and PKE-IK-CCA secure, then Π^2_{PKE} in Figure 3.3 is secure and strongly MITM-private.*

Proof. We prove the properties separately.

Π^2_{PKE} **is secure.** Let π_i^s be the tested oracle. Let $j = \mathsf{Pid}_\mathsf{i}^\mathsf{s}$. π_i^s must conform to freshness (Definition 11). **Case (a)** Clause 3a was fulfilled, i.e. there is a partner oracle π_j^t. It follows that π_j^t has matching conversations to π_i^s. **Case (b)** Clause 3b was fulfilled, which means j must not be corrupted. Together with the fact that π_i^s accepted, from Lemma 11 follows that there is some π_j^t that has matching conversations to π_i^s. It follows that in any case there is some π_j^t that has matching conversations to π_i^s.

To distinguish the session key k from random, $\mathcal{A}$ needs to query $H(\Gamma.key, x, \mathsf{ctxt}_3)$. Following the proof of Lemma 11, we can again use game hops that replace c_1 with the ciphertext of some random value to show that no information about x is leaked by c_1. Since x is otherwise only used as input for the RO, $\mathcal{A}$ only has $\mathsf{negl}(\lambda)$ chance to win the game (e.g. by guessing x or sk_j).

Π^2_{PKE} **is strongly MITM-private.** We distinguish cases based on who is the initiator.

- **Case 1.** $\pi^1_{i|j}$ is $\mathsf{Initiator}$.
  Therefore $k := \mathsf{Pid}^1_{\mathsf{i|j}}$ is immediately set.
  - Game 0: The original game.
  - Game 1: Guess i, j, k. Abort if $\mathsf{Test}(m)$ does not return $i|j$ or $k \neq \mathsf{Pid}^1_{\mathsf{i|j}}$ at the end of the game.
  - Game 2: Replace m_1 by $\pi^1_{i|j}$ with $m_1^* = \mathsf{PEnc}_k(z)$ where z is random bit string. Program all other oracles to treat m_1^* as m_1 (i.e., the decryption is x).

  The transitions are the same as in Lemma 11 hence we skip them here. Notice that in Game 2 $\mathcal{A}$ can only guess, hence the probability of winning Game 2 is $\frac{1}{2}$.

- **Case 2.** $\pi^1_{i|j}$ is $\mathsf{Responder}$. m_2 does not depend $\mathsf{pk}_{i|j}$ and is sent even after receiving messages that are invalid or cannot be decrypted. Below we argue why m_1 does not reveal the key that was used for encryption. Since PKE is PKE-IK-CCA, we can replace $\mathsf{pk}_{i|j}$ with a random key. To show this, consider the game hops below. Note that a valid or invalid m_2, i.e. $\pi^1_{i|j}$ being able to decrypt m_1 or not, does not give any information about the secret bit b anymore if $\mathsf{pk}_{i|j}$ is replaced with a random key.
  - Game 0: The original game.
  - Game 1: Whenever any oracle is instructed to initiate communications with $\pi^1_{i|j}$, it saves the message it produced, i.e. m_1, in a secret table together with the data that was encrypted. $\pi^1_{i|j}$, instead of decrypting incoming messages, will look up the content in the secret table. If the message is not in the table, it will attempt to decrypt the message normally.
  - Game 2: $\pi^1_{i|j}$ treats the incoming message m_1 as malformed if there is no matching entry in the secret table. (Note that in this game, under no circumstances $\pi^1_{i|j}$ actually decrypts any messages.)

– Game 3: Instead of using the public key of i or j (depending on the secret bit b), the party $P_{i|j}$'s public key is set to a randomly generated key pk'.

The game hopes are based on the same indistinguishable game hops as discussed in Theorem 2, hence we do not repeat them here.

□

Lemma 13. *If PKE* PKE *is PKE-IK-CCA secure, then* Π^2_{PKE} *is completed-session private.*

Proof. Due to PKE-IND-CCA-security of PKE, m_1 and m_2 can be replaced with encryptions of random content in this proof (neither party may be corrupted). Due to PKE-IK-CCA security of PKE the used keys can be replaced with random keys (similar to proof of Theorem 2). It follows that the full transcript is randomly generated, i.e. independent of secret bit b. □

3.2.3 One-move PPAKE

The question whether a 1-move PPAKE protocol can provide desireable privacy guarantees is left as an open question for future research. It might not be as far out of reach as it seems, if powerful (and costly) primitives such as puncturable encryption (PE) [GM15, DGJ+21] are considered.

A potential protocol might have the initiator send a random nonce as well as its own identity and signature to the recipient using a PE. This protocol would hence correspond to the first move of the protocol Π^2_{PKE}. The recipient would then decrypt the message and puncture its secret key for that message, effectively making it impossible to use the secret key to later decrypt the message again. This effectively provides forward secrecy. The session key could be derived from hashing the identities and the nonce.

However the above approach has the problem that for the primitives we examined, leaking the secret key reveals which messages the key was punctured for. Specifically, if an adversary intercepts a message m (without knowing its recipient) and later corrupts some potential recipients' secret keys, they are then able to determine if one of the potential recipients have decrypted m.

Another approach is using puncturable encryption, but instead of puncturing for the message after it was received, puncture for a time period after it ends. However to formally analyze such a protocol, significant changes have to be made to our model in order to incorporate time periods and security notions that rely on them. Furthermore the practical usefulness is not clear, since the involved primitives are so costly. Hence this topic is left to future research.

3.3 Existing Protocols in the Literature

In this section we will discuss two protocols from recent papers (by Zhao [Zha16] and Schäge, Schwenk and Lauer [SSL20]) on PPAKE protocols (also called identity-conceiling AKE protocols in the literature). As we will show, both fulfill completed session privacy but not MITM privacy.

3.3.1 Construction by Zhao

Among other contributions, Zhao [Zha16] presents the protocol CAKE that aims to authenticate and protect the privacy of both parties.

While Zhao [Zha16] uses a notably different model[3], the protocol CAKE in Figure 3.4 can be evaluated in our model without any modifications.

Theorem 5. *Protocol CAKE in Figure 3.4 is secure and completed session private under AEAD security [4] of (E, D) and the GDH assumption in the random oracle model.*

Completed Session Privacy [Recap]
Goal: Return bit b, indicating wether the new party $P_{i\|j}$ is equivalent to P_i or P_j.
Restrictions: The returned oracle $\pi^1_{i\|j}$'s state is Accept at the end of the game. Let $k = \mathsf{Pid}^1_{\mathsf{i\|j}}$. P_k are not corrupted. Also $\mathsf{RevSessKey}(i\|j, 1)$ was never queried and $\mathsf{RevSessKey}(k, r)$ (for any π^r_k that has matching conversations) was never queried.

Proof (Sketch). The fact that protocol CAKE is "strongly CAKE-secure" in Zhao's model was proven in [Zha16]. The fact that this relates to completed session privacy is left to the reader's inspection.

Lemma 14. *Protocol CAKE in Figure 3.4 is not MITM private.*

[3]Zhao's model [Zha16] does not inform the initiator about the intended recipient. As a consequence, there is no need to deal with informing other oracles about the test oracle's identity, and the model hence easily avoids potential trivial attacks (c.f. the attack discussed in Section 2.5.5). On the other hand this makes is impossible to e.g. evaluate Π_{Gen} in Zhao's model.

[4]AEAD security is not formally introduced in this work. Refer to [Zha16] for formal definition. For understanding the protocol CAKE, AEAD security can be seen as identical to LH-SE-IND-CCA-security.

Alice	**Bob**
pid_A	pid_B
$PK_A : A = g^a$	$PK_B : B = g^b$
$SK_A : a \leftarrow Z_q^*$	$SK_B : b \leftarrow Z_q^*$
$X \leftarrow g^x$	
$d = h(X, pid_A)$	
$\xrightarrow{\overline{X} = AX^d}$	
	$Y = g^y$
	$e = h(\overline{X}, Y, pid_B)$
	$PS = \overline{X}^{b+ye}$
	$(K_1, K_2) = \mathsf{KDF}(PS, \overline{X}\vert\vert\overline{Y})$
	$\xleftarrow{H_B, \overline{Y} = BY^e, C_B = E_{K_1}(H_B, Y\vert\vert pid_B)}$
$PS = \overline{Y}^{a+xd}$	
$(K_1, K_2) = \mathsf{KDF}(PS, \overline{X}\vert\vert\overline{Y})$	
$D_{K_1}(H_B, C_B) = (pid_B, Y)$	
Compute $e = h(\overline{X}, Y, pid_B)$	
Check whether $\overline{Y} = BY^e$, and abort if not	
Session-key is set to be K_2	
$\xrightarrow{H_A, C_A = E_{K_1}(H_A, pid_A\vert\vert X)}$	
	$D_{K_1}(H_A, C_A) = (A, X)$
	Compute $d = h(X, pid_A)$
	Check whether $\overline{X} = AX^d$, and abort if not
	Session-key is set to be K_2

Figure 3.4: Protocol CAKE, see Fig. 7 by Zhao [Zha16], using AEAD $\Omega = (E, D)$ [1] and key-derivation-function KDF[2]

[1] Authenticated Encryption with Associated Data (AEAD) is not formally introduced in this work. For understanding the protocol, AEAD can be seen as equivalent to symmetric encryption. Refer to [Zha16] for a formal definition of AEAD.
[2] Similarly, key-derivation-function can be viewed as hash functions for understanding the protocol.

weak MITM Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: No oracle is ever corrupted.

Proof (Sketch). Note that the initiator only reveals its identity in the third message of the protocol. Moreover, any adversary can generate some $a \leftarrow Z_q^*, A = g^a$. This means an adversary can set $pid_A = 0$ for purposes of computing $d = h(X, pid_A)$ and run the protocol normally, but abort instead of sending the third message. This is sufficient to deanonymize the responder, as it sent pid_B in ciphertext which the adversary can decrypt.

Lemma 15. *Protocol CAKE in Figure 3.4 is forward private under AEAD security of (E, D) and the GDH assumption in the random oracle model.*

Forward Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: The returned oracle $\pi^1_{i|j}$ has a partner oracle π^r_k at the end of the game. Furthermore no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i|j}$.

Proof (Sketch). x and y are produced by the partner oracles, and it is hence hard for the adversary to compute $PS = \overline{X}^{b+ye} = \overline{Y}^{a+xd}$ which is needed for the input to KDF. The proof therefore follows a similar argument as for the forward privacy of Π_{ss}.

3.3.2 Construction by Schäge, Schwenk and Lauer

Schäge, Schwenk and Lauer [SSL20] examine the privacy of IKEv2. In their paper they detail the full version of IPsec IKEv2 Phase 1 with digital signature based authentication. Furthermore they show a simplified version, that only incorporates the security and privacy relevant aspects. For brevity, and since both versions indeed fulfill the same privacy properties in our model, we show only the simplified version.

Schäge, Schwenk and Lauer also use a different model [5], but Π_{SSL} in Figure 3.5 can again be evaluated in our model without any change. For the purposes of our model, ID_A and ID_B simply relate to the party's index, instead of some identity selector bit like in the model of Schäge, Schwenk and Lauer.

Theorem 6. *Protocol Π_{SSL} in Figure 3.5 is secure and completed session private, if KDF, $\Omega = (E, D)$ and auth are secure as defined in [SSL20].*

[5] The model of Schäge, Schwenk and Lauer [SSL20] gives all parties two identities, and the adversary must discern between the two identities of any party it chooses. In order to evaluate Π_{Gen} in their model, we need to consider their model in the mode that the initiator chooses the responder's ID.

Alice		**Bob**
(sk_A, pk_A)		(sk_B, pk_B)
	Key Agreement	
	$\xrightarrow{m_1 = g^x}$	
	$\xleftarrow{m_2 = g^y}$	
	Key Derivation $k \leftarrow KDF(g^{xy})$	
	Apply Encryption to Identity-Related Data	
	$\xrightarrow{m_3 = E_k(ID_A, pk_A, auth_A)}$	
		if $auth_A$ invalid abort
	$\xleftarrow{m_4 = E_k(ID_B, pk_B, auth_B)}$	
if $auth_B$ invalid abort		

Figure 3.5: Protocol Π_{SSL}, c.f. Fig. 2 by Schäge, Schwenk and Lauer [SSL20]

Completed Session Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: The returned oracle $\pi^1_{i|j}$'s state is Accept at the end of the game. Let $k = \mathsf{Pid}^1_{\mathsf{i|j}}$. P_k are not corrupted. Also $\mathsf{RevSessKey}(i|j, 1)$ was never queried and $\mathsf{RevSessKey}(k, r)$ (for any π^r_k that has matching conversations) was never queried.

Proof (Sketch). The security and privacy of Π_{SSL} was shown in [SSL20]. The fact that this relates to completed session privacy in our model is left to the reader's inspection.

Lemma 16. *Protocol Π_{SSL} in Figure 3.5 is not MITM private.*

weak MITM Privacy [Recap]
Goal: Return bit b, indicating wether the new party $P_{i\|j}$ is equivalent to P_i or P_j.
Restrictions: No oracle is ever corrupted.

Proof (Sketch). Note that the responder only reveals its identity in the fourth message of the protocol. This means an adversary (even without key material) may run the protocol normally, but abort instead of sending the fourth message. This is sufficient to deanonymize the initiator, who authenticates itself in the third message.

As mentioned before, this attack was already noted by Schäge, Schwenk and Lauer.

Lemma 17. *Protocol Π_{SSL} in Figure 3.5 is forward private if KDF is instantiated as RO H and (E, D) is a LH-SE-IND-CCA secure symmetric encryption scheme.*

Forward Privacy [Recap]
Goal: Return bit b, indicating wether the new party $P_{i\|j}$ is equivalent to P_i or P_j.
Restrictions: The returned oracle $\pi^1_{i\|j}$ has a partner oracle π^r_k at the end of the game. Furthermore no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i\|j}$.

Proof (Sketch). This lemma follows from the hardness of computing g^{xy} as well as the security of E, D (refer to the forward privacy proof of Π_{Gen} in Section 3.1.5).

3.4 Summary

In this chapter we showed several protocols that fulfill different levels of privacy guarantees. This is summarized in Table 3.1.

It is possible that even a 1-move PPAKE with strong MITM privacy and forward privacy is feasible. This could potentially be constructed from puncturable public encryption schemes. However the practical usefulness is unclear due to the high cost of this primitive. Furthermore it requires some modifications to the model. For these reasons this question is left open for future work.

We also discussed two protocols of the literature, which both fulfill completed session privacy and forward privacy. However they do not provide MITM privacy (see Lemma 14).

Table 3.1: Comparison of our protocols. "ss" denotes the requirement of a shared secret and "pk" the requirement to know the public key of the intended responder upfront.

	ss	pk	forward priv.	comp.-ses. priv.	w.-MITM	s.-MITM	# moves
Π_{Gen}	×	✓	✓	✓	✓	✓	4
Π_{ss}	✓	×	✓	✓	✓	×	3
Π^2_{PKE}	×	✓	×	✓	✓	✓	2

The corresponding type of attack was noted by the authors, but purposefully excluded from their models. Even before, Krawczyk [Kra03] mentioned this attack and called it unavoidable, as one party must always "go first" with authenticating. However this is only true if we assume that neither party knows their communication partner beforehand. As was shown with the protocols Π_{Gen} and those in Section 3.2, this problem can be overcome if the initiator "go first", but uses the intended recipients public key for encrypting its own identity. We believe that the requirement of the initiator knowing who they plan on contacting is reasonable for most real-world application scenarios. The fact that the public key must be known beforehand of course requires the public keys to be shared beforehand or there must be a method of obtaining the public key, similar to DNS requests in the Internet setting. This method must clearly be secure and potentially privacy preserving as well.

CHAPTER 4

Automated Verification

4.1 Overview

In this chapter we discuss how automated verification can be used to prove the privacy of Π_{Gen} (introduced in Section 3.1). For this purpose we first present the code for a Tamarin Prover representation that encodes the protocol and our model. However, this encoding turns out to be not successful, since Tamarin is not able to finish its computations (Out-Of-Memory exception even when supplied with approximately 60 GB of RAM). Hence we created a second encoding, in this case for ProVerif, which allows us to successfully prove our desired result.

Tamarin Prover vs. ProVerif. As mentioned above, our Tamarin Prover encoding did not work. This yields the question, whether there might be a different encoding that works. To answer this, consider the following: The main purpose of automated verification is to give additional confidence in the security of cryptographic constructions. In order to accomplish this, the encoding has to closely resemble the actual constructions and security properties. Due to limitations of the respective tool's language, one usually can only encode protocols and properties that are similar or optimally even equivalent to the original. This might also entail using unnatural workarounds.

Hence, to answer the initial question, yes, there surely is some Tamarin encoding that can be successfully proven, since the underlying statement "the protocol is secure" is true (as proven classically and with ProVerif). However that encoding probably has no value for giving additional confidence, as it probably requires a large amount of workarounds and modifications to the protocol.

For this reason we chose to instead utilize ProVerif, in particular since it allows us to naturally encode if/then/else-clauses, which are needed to model Π_{Gen}. Specifically, in Π_{Gen} the responder performs several checks before sending either a correctly formed

m_4 or a random value. Tamarin does not have if/then/else in its language and there is also no simple workaround (see Section 4.2.1). ProVerif however allows if/then/else statements inside of terms. This has two significant advantages. First of all it is easier to see that our ProVerif encoding actually represents our protocol, than it is with the workaround we had to use in the Tamarin encoding. Secondly, this native language construct in ProVerif is more efficient than our workaround in Tamarin.

Benchmarks. The encodings that will be shown in this chapter have been executed on a Windows PC using the WSL (Windows Subsystem for Linux) with Ubuntu 20.04.1 LTS. The PC contains an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and has 32 GB of RAM installed. This setup led to the following runtimes:

- Π_{Gen} Tamarin formulation for strong MITM privacy: Out-Of-Memory error after approximately 2 hours[1]
- Π_{Gen} ProVerif formulation for strong MITM privacy: 53 minutes
- Π_{Gen} ProVerif formulation for forward privacy: 10 minutes

4.2 Tamarin Prover

In this section we discuss how we encoded Π_{Gen} for the Tamarin Prover in our attempt to prove strong MITM privacy. Recall that in Tamarin we use *builtins*, custom *functions* and *equations* to define the building blocks of our protocol. The protocol itself is specified using *rules* and our security properties are encoded using *restrictions* and *lemmas*. (See Section 1.3.)

4.2.1 Modelling if/else for observational equivalence

The main challenge when modelling Π_{Gen} was the lack of if/else-constructs in the Tamarin Prover's language, which would be needed to implement the responder's behavior for sending the real or random m_4. Using multiple rules for the different execution paths of if/else does not work with observational equivalence, which requires that the same rules are applied in both worlds in the same order. As a workaround, we used Tamarin's equations. This is possible in this scenario, as Π_{Gen} only needs if/then to have the responder either send a real m_4, if all checks succeeded, or send a random value otherwise.

Specifically, in place of m_4 the responder sends a term that is a function of the actual m_4, a random nonce and the verification checks. We define an equation that sets a term of this form equal to the actual m_4, but only if the verification results in *true*. This is exemplified with the code snippet below, which is a simplified part of the code for Π_{Gen} that is shown later.

[1] We also executed the encoding in a virtual machine running a computer cluster. We allocated 64 GB of RAM, but still ran out of memory.

```
functions: CalcM4/4
equations: CalcM4(m, r, true, true) = m

// ...

// Using it in a rule:
rule SendM4:
let
// ... (definitions of sig, msg, pk1, sig2, msg2, pk2, realM4)
in
[ Fr(~rand),
    //...
] -->
[ Out(CalcM4(realM4, ~rand, verify(sig, msg, pk1), verify(sig2, msg2, pk2))) ]
```

Listing 4.1: Conditionally revealing information

In Listing 4.1 we use the function *CalcM4* to either hide or reveal the real m_4. The first parameter m is filled with the real m_4, the second parameter r is filled with a fresh (random) value to make it impossible for the adversary to recompute the term. The final two parameters are for the verification checks, that are carried out by the responder in Π_{Gen} before sending m_4.

Note that the builtin verification checks are realized with their own equational theory, and are per definition equal to true in case that the signature (the first parameter) matches the specified message (the second parameter) and public key (the third parameter). Otherwise they remain "not simplified". Similarly, our term that consists of *CalcM4* stays some unknown term, that cannot be recomputed by the adversary (due to *~rand*), unless the verification checks succeeded, in which case it can be simplified to *realM4*. In other words we approximated outputting a random value in case that the verifications fail (as done in the real protocol), by outputting some unknown value in the form of a "not simplified" term.

This method is successful in modelling our goal and works for proving the lemmas we specified (some sanity checks and a proof of explicit authentication). However, it leads to a high computational effort, which makes observational equivalence unprovable on a reasonably powerful machine, as discussed before.

4.2.2 Π_{Gen} Formulation (Tamarin Prover)

Below we give the full encoding of proving strong MITM privacy for Π_{Gen}. The Tamarin Prover eventually encountered an Out-Of-Memory exception and was hence unable to complete its proof or find an attack (even for versions in which some aspects like the adversary oracles were removed). Since we therefore switched to ProVerif, we omit discussing the adequacy of the Tamarin encoding and only list it for reference.

```

theory PiGen
begin

builtins: hashing, asymmetric-encryption, signing, symmetric-encryption

functions: KE_m1/1, KE_m2/2, KE_kA/2, KE_kB/2, CalcM4/4
equations: KE_kA(randA, KE_m2(randB, KE_m1(randA))) = KE_kB(randB, KE_m1(randA)),
           CalcM4(m, r, true, true) = m

// Create the CA
rule CA_Init:
    [ Fr(~ltk) ]
    --[CA_Init()]->
    [ !CA(~ltk, pk(~ltk)) ]

restriction CA_Init_Once:
"
    All #i #j . CA_Init() @ #i & CA_Init() @ #j ==> #i = #j
"

// This corresponds to the CA signing some pks
rule CreateIdentity:
    let
        caSig = sign(<~id, pk(~ltk_Sign)>, ltk)
    in
    [ Fr(~id), Fr(~ltk_Sign), Fr(~ltk_AEnc), !CA(ltk, pub) ]
    --[CreatedParty(~id, ~ltk_Sign, ~ltk_AEnc)]->
    [ !Party(~id, ~ltk_Sign, ~ltk_AEnc, caSig),
        Out(~id), Out(pk(~ltk_Sign)), Out(pk(~ltk_AEnc)), Out(caSig) ]

// Session ID could also correspond to connection endpoint, i.e. Port in IP Internet model
rule InitializeProtocol:
    let
        m1Inner = KE_m1(~rand)
        m1 = <~chnl, m1Inner>
```

```
    in
    [ !Party(a, aSign, aEnc, caSig), Fr(~chnl), !Party(b, bSign, bEnc, caSig), Fr(~rand) ]
   --[M1(a, ~chnl, m1), Peer(a, b)]->
    [ Out(m1), M1Done(a, ~chnl, b, ~rand, m1Inner), Randomness(a, ~rand) ]

rule SendM2:
    let
        m1_Rec = <chnl, m1>
        m2Inner = KE_m2(~rand, m1)
        m2 = <chnl, m2Inner>
        ctxt = <m1, m2Inner>
    in
    [ Fr(~rand), !Party(b, bSign, bEnc, caSig), In(m1_Rec) ]
   --[M2(b, chnl, m2)]->
    [ Out(m2), M2Done(b, chnl, KE_kB(~rand, m1), m1, m2Inner), Randomness(b, ~rand) ]

rule SendM3:
    let
        m2_Rec = <chnl, m2>
        ctxt = <m1, m2>
        KE_k = KE_kA(rand, m2)
        k = h(<KE_k, ~x, ctxt>)
        m3Inner = <senc(aenc(~x, pk(bEnc)), KE_k), senc(<a, pk(aSign), caSig, sign(ctxt, aSign)>, k)>
        m3 = <chnl, m3Inner>
    in
    [ Fr(~x), In(m2_Rec), M1Done(a, chnl, b, rand, m1), !Party(a, aSign, aEnc, caSig), !Party(b, bSign, bEnc, caSigB) ]
   --[M3(a, chnl, m1, m2, m3Inner, k)]->
    [ Out(m3), M3Done(a, chnl, b, ~x, m1, m2, m3Inner, k) ]

rule SendM4:
    let
        ctxt = <m1, m2>
        m3 = <c0, c1>
        m3_Rec = <chnl, m3>
        x = adec(sdec(c0, KE_k), bEnc)
        k = h(KE_k, x, ctxt)
        c1Dec = sdec(c1, k)
        a = fst(c1Dec)
```

```
        pkA = fst(snd(c1Dec))
        caSig = fst(snd(snd(c1Dec)))
        aSig = snd(snd(snd(c1Dec)))
        ctxt2 = <a, b, m1, m2, m3>
        m4Inner = <x, ctxt2>
        yesOrNo = CalcM4(h(m4Inner), ~rand, verify(caSig, <a, pkA>, pub), verify(aSig, ctxt, pkA))
        m4 = <chnl, yesOrNo>
        ctxt3 = <a, b, m1, m2, m3, yesOrNo>
        kResult = h(k, x, ctxt3)
    in
    [ In(m3_Rec), M2Done(b, chnl, KE_k, m1, m2), Fr(~rand), !CA(ltk, pub), !Party(b, bSign, bEnc, caSigB) ]
    --[B_Finished(b, chnl, ctxt2, h(m4Inner), yesOrNo, kResult), M4Done(ctxt3), Peer(b, a)]-> //If h(m4Inner) = yesOrNo then this accepts (since this means the verifys are true)
    [ Out(m4), KeyDerived(kResult, a, b), Randomness(b, ~rand) ]

rule ReceiveM4:
    let
        m4 = <chnl, hash>
        ctxt3 = <a, b, m1, m2, m3, hash>
        kResult = h(k, x, ctxt3)
    in
    [ In(m4), M3Done(a, chnl, b, x, m1, m2, m3, k) ]
    --[A_Finished(a, chnl, ctxt3, h(<x, <a, b, m1, m2, m3>>), hash, kResult)]-> //If h(...) = hash then this accepts
    [ KeyDerived(kResult, a, b) ]

// ------ Test queries ---------
// Privacy
rule TestPrivacy:
    [ !Party(a, k1, k2, caA), !Party(b, l1, l2, caB), Fr(~c), !CA(sk, pk) ] --[ Test(), TestPriv(a, b, ~c) ]->
    [ !Party(~c, diff(k1, l1), diff(k2, l2), sign(<~c, diff(k1, l1)>, sk)), Out(~c) ]

// Key Indistinguishiability
rule TestKeyInd:
    [ KeyDerived(k, a, b), Fr(~r) ] --[ Test(), TestKeyInd(a, b) ]-> [ Out(diff(k, ~r)) ]

```

```
restriction SingleTest:
"
    All #i #j . Test() @ #i & Test() @ #j ==> #i = #j
"

// ------ Oracles -------

rule ORevLTK:
    [ !Party(a, k1, k2, caA) ] --[Corrupted(a)]-> [ Out(k1), Out(k2) ]

rule ORegisterLTK:
    let
        caSig = sign(<~id, pkAdv>, ltk)
    in
    [ Fr(~id), In(pkAdv), !CA(ltk, pub) ] --[Corrupted(~id)]-> [ Out(caSig) ]

rule ORevSessKey:
    [ KeyDerived(k, a, b) ] --[ Revealed(a, b) ]-> [ Out(k) ]

// Setting
restriction TestKeyInd_Fresh:
"
    All a b #i . TestKeyInd(a, b) @ #i ==> not(Ex #j . Revealed(a, b) @ #j)
"

restriction Test_s_MITPriv:
"
    All a b c #i . TestPriv(a, b, c) @ #i ==>
        not (Ex #j . Corrupted(c) @ #j)
        & (All x #l . Peer(c, x) @ #l ==> not (Ex #m . Corrupted(x) @ #m))
        & not (Ex #j . Corrupted(a) @ #j)
        & not (Ex #j . Corrupted(b) @ #j)
"

// ------ Lemmas --------
// Exists trace
lemma Satisfiable_AcceptingSameKey:
exists-trace
" Ex a sessA b sessB ctxt2 ctxt3 hash msg kResult #i #j .
```

```
        A_Finished(a, sessA, ctxt3, hash, hash, kResult) @ #i & B_Finished(b, sessB, ctxt2, msg, msg, kResult) @ #j "

// Explicit authentication
lemma ExplicitAuthentication_BFinished:
"
    All b sess ctxt2 msg kResult #i . B_Finished(b, sess, ctxt2, msg, msg, kResult) @ #i ==>
        (Ex a sess2 m1 m2 m3Inner k #j . M3(a, sess2, m1, m2, m3Inner, k) @ #j & ctxt2 = <a, b, m1, m2, m3Inner>)
"

lemma ExplicitAuthentication_AFinished:
"
    All a sessA ctxt3 hash kResult #i . A_Finished(a, sessA, ctxt3, hash, hash, kResult) @ #i ==> (Ex #j . M4Done(ctxt3) @ #j & #j < #i)
"

end
```

Listing 4.2: s-MITM Privacy of Π_{Gen} (Tamarin Prover)

4.3 ProVerif

In this section we present our encoding of the Π_{Gen} protocol in the setting of either strong MITM privacy or forward privacy.

4.3.1 Introduction

We utilize ProVerif's process format (recall Section 1.4). This means we first introduce several types, functions and equations, define our queries and then specify the processes. Note that the queries cannot be written in first order logic, but rather can only have one of a few specific forms. In our formulation we use observational equivalence (for the main proof) as well as reachability and implication queries (for sanity checking). Since ProVerif only allows either observational equivalence terms or queries, the latter will be commented in the code that is shown later. We now first discuss our formulation of Π_{Gen} and strong MITM privacy, including the simplifications that we have made to the model specified in Section 2.3. Afterwards we show how the encoding can be adapted to prove forward privacy.

4.3.2 Modified model

Due to ProVerif's modelling limitations, we have to make the following adaptions to the model.

A Priori Corruptions. In our model (see Section 2.3) the adversary is allowed to call $\mathsf{RevLTK}()$ to dynamically corrupt oracles. Trying to model this in ProVerif comes with difficulties. Note that for strong MITM privacy, the adversary might lose the game by corrupting the peer of the test oracle, after their communication is completed. This cannot be modelled directly, as there is no "the adversary has lost now" command in ProVerif. The above case could be mitigated with a workaround (i.e. refusing to carry out the corruption in this scenario), but there are more scenarios that make the adversary lose based the same restriction, like corrupting the peer before the communication.

In order to avoid these problems, we simplified the model in a way that should be equivalent to the original model (i.e. allow or prevent the same attacks). We model corruptions by grouping the parties into two arbitrarily large disjunct sets, one with honest parties and one with corrupted parties. Essentially this means that the adversary must decide a priori which parties it will corrupt later. This change seems to be reasonable, considering that strong MITM privacy in our original model is already not influenced by the specific time at which the adversary corrupts some oracle (recap below).

strong MITM Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: P_i and P_j are never corrupted. Furthermore we require that $\mathsf{Pid}^1_{\mathsf{i|j}} = \emptyset$ or $\mathsf{Pid}^1_{\mathsf{i|j}} = k$ for some k, while P_k is never corrupted.

Dummy messages instead of aborting. Note that in our original model, the adversary loses the game if $\mathsf{Pid}^1_{\mathsf{i|j}} = k$ and k is corrupted at any point. In ProVerif, if the adversary first corrupts k, then it is difficult to prevent them from utilizing this information to identify themselves to the responder. It is easy to detect that $\mathsf{Pid}^1_{\mathsf{i|j}} = k$ occurs, but ProVerif, again, does not offer a "the adversary has lost now" command. Instead the experiment must continue, and we have to make sure that the adversary cannot possibly win now. In this case the responder should send m_4, but instead sends a random value. Depending on the attack scenario, the adversary can either not detect this change or it does not provide any additional information to them.

Fixed test party. In our original model, the adversary is able to dynamically choose which parties it wants to distinguish. Since all parties are equivalent in our ProVerif formulation, we fix two parties, A and B, as well as the test party AB which has the same keys as either A or B.

Forward privacy. In our model, forward privacy requires the existence of a partner oracle (see recap below).

Forward Privacy [Recap]

Goal: Return bit b, indicating wether the new party $P_{i|j}$ is equivalent to P_i or P_j.

Restrictions: The returned oracle $\pi^1_{i|j}$ has a partner oracle π^r_k at the end of the game. Furthermore no oracle besides π^r_k may be instructed to start a protocol run with intended partner $P_{i|j}$.

This implies that both oracles agree on the transcript, which means that the adversary did not change any messages between these two oracles. In ProVerif, we realize this by setting the adversary mode to *passive*, which makes the adversary entirely unable to create its own messages. Formally, this is a stronger restriction than in our model, since our model allows the adversary to create messages and send them to unrelated oracles. However we argue that communicating with an unrelated oracle (that does not intend to contact the test oracle, as per restriction in the model) does not benefit the attacker.

4.3.3 Π_{Gen} sMITM Privacy Formulation (ProVerif)

Code Overview

In this section we list the different segments of the code, which will be listed fully in the next section. Furthermore we discuss the goals and notable design choices of each segment.

Basics. In this segment we introduce the cryptographic primitives that are independent of the current protocol. The part "key exchange" correlates to primitives like the Diffie-Hellman key exchange, as discussed in Section 1.2. Notably, we use equations to model e.g. the behavior of decryptions, instead of using the *reduc* keyword. This is necessary because otherwise decryption failures (in particular of the responder after receiving m_3) cause observational equivalence to be unprovable.

General. This segment contains all remaining definitions.

- Names A, B and AB are for the fixed test party (see Section 4.3.2).
- Table t is used by the initiator to retrieve the responder's public keys.

- We model the certificates issued by the Certificate Authority (CA) as a private function because we need a way to check whether a party name and public key is valid, without potentially causing errors[2].
- Party status is modelled as part of the name, since, as discussed in Section 4.3.2, parties are honest or corrupt from the start. Furthermore we again needed a way to check the party status, that does not cause an error (or different execution path).
- For the latter reason, we also introduced tuple deconstruction helpers that never fail, even though tuple deconstruction can be done with native language constructs.
- The private function *createK* is used to replicate the behavior of RevSessKey() in our model in case that the oracle did not accept (i.e. output random keys, but output the same key if the context is the same to a previously queried one, c.f. Section 2.3.3).
- Finally there are some events, which are irrelevant for the main observational equivalence proof. They are only used for sanity checking via reachability queries.

Protocol. As the name suggests, this segment contains the protocol definition. Contrary to the Tamarin Prover formulation, we do not need *CalcM4* functions and equations, since ProVerif allows us to specify terms that contain *if/then/else*.

Full Code

Below we list the full ProVerif code for showing strong MITM privacy. It was successfully proven (i.e. observational equivalence was proven).

```

(* ######## Basics ######### *)
(* Symm Encr *)
type key.
fun senc(bitstring, key): bitstring.
fun sdec(bitstring, key): bitstring.
equation forall m: bitstring, k:key; sdec(senc(m,k),k) = m.
fun bitToKey(bitstring) : key [typeConverter].

(* Asymm Encr *)
type skey.
type pkey.
fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
```

[2] If, for example, we created a table of all valid parties, then looking up one party's information would fail in case that the adversary provided invalid data. This would prevent observational equivalence to be proven, even though the observables in our model (i.e. the sent message) would be indistinguishable.

```
fun adec(bitstring, skey): bitstring.
equation forall m: bitstring, k: skey; adec(aenc(m, pk(k)), k)
   = m.

(* Signatures *)
type sskey.
type spkey.
type sigValidity.
const sigVALID : sigValidity.
fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
fun vfySig(bitstring, bitstring, spkey) : sigValidity.
reduc forall m: bitstring, k: sskey; getmess(sign(m, k)) = m.
equation forall m: bitstring, k: sskey; vfySig(m, sign(m,k),
   spk(k)) = sigVALID.

(* Randomness *)
type randomness.

(* Hash *)
fun h(bitstring): bitstring.

(* Key Exchange *)
type ke_m1.
type ke_m2.
fun ke1(randomness) : ke_m1.
fun ke2(randomness, ke_m1) : ke_m2.
fun keK1(randomness, ke_m2) : bitstring.
fun keK2(randomness, ke_m1) : bitstring.
equation forall r1, r2 : randomness; keK1(r1, ke2(r2, ke1(r1)))
   = keK2(r2, ke1(r1)).

(* ####### General ######### *)
channel c.

type name. (* Identify parties *)
const A, B, AB : name.
table t(name, pkey, spkey). (* Table for retrieving pks *)

(* Certificates *)
type caCertValidity.
const caVALID: caCertValidity.
fun caCert(name, spkey) : bitstring [private].
```

```
fun caCertName(bitstring) : name.
fun caCertKey(bitstring) : spkey.
fun caCertValid(bitstring) : caCertValidity.
equation forall n: name, k: spkey; caCertName(caCert(n, k)) = n.
equation forall n: name, k: spkey; caCertKey(caCert(n, k)) = k.
equation forall n: name, k: spkey; caCertValid(caCert(n, k)) = caVALID.

(* Party Status *)
type partyStatus.
const psHONEST, psCORRUPT : partyStatus.
fun psName(name, partyStatus):name [private].
fun isPartyOk(name):partyStatus.
equation forall n: name, ps: partyStatus; isPartyOk(psName(n, ps)) = ps.

(* Tuple Deconstructor Helper *)
fun p1(bitstring): bitstring.
fun p2(bitstring): bitstring.
equation forall x : bitstring, y : bitstring; p1((x, y)) = x.
equation forall x : bitstring, y : bitstring; p2((x, y)) = y.

(* Random Key Generation by RevSessKey Oracle *)
fun createK(ke_m1, ke_m2, bitstring, bitstring) : bitstring [private].

(* Events *)
(* event initAccepts(k, m1, m2, m3, m4, expectedM4). *)
event initiatorDone(bitstring, ke_m1, ke_m2, bitstring, bitstring, bitstring, partyStatus).

(* event responderDone(k, m1, m2, m3, m4, caVal, vfySig, ps). *)
event responderStarted(name).
event responderDone(name, bitstring, ke_m1, ke_m2, bitstring, bitstring, caCertValidity, sigValidity, partyStatus).

(* ####### Protocol ######## *)
let Initiator(name: name, sgn : sskey) =
    (* Initialize *)
    in(c, peer : name); (* Receive adversary instruction of whom to contact. *)
```

```
    if name <> psName(AB, psHONEST) || isPartyOk(peer) =
psHONEST then (* Party AB only talks to honest parties *)
    get t(=peer, bPKE : pkey, bSgn : spkey) in

    (* m1 *)
    new rand : randomness;
    let m1 = ke1(rand) in
    out(c, m1);

    (* rcv m2 & calc key *)
    in(c, m2 : ke_m2);
    let kek = bitToKey(keK1(rand, m2)) in

    (* send m3 *)
    new x : bitstring;
    let ctxt = (m1, m2) in
    let k' = bitToKey(h((kek, x, ctxt))) in
    let c0 = senc(aenc(x, bPKE), kek) in
    let c1 = senc((caCert(name, spk(sgn)), sign((name, peer, c0
, ctxt), sgn)), k') in
    let m3 = (c0, c1) in
    out(c, m3);

    (* rcv m4 *)
    in(c, m4 : bitstring);
    let ctxt2 = (name, peer, m1, m2, m3) in
    let ctxt3 = (name, peer, m1, m2, m3, m4) in
    let expectedM4 = h((x, ctxt2)) in
    let k = h((kek, x, ctxt3)) in
    event initiatorDone(k, m1, m2, m3, m4, expectedM4,
isPartyOk(peer));

    out(c, if m4 = expectedM4 then k else createK(m1, m2, m3,
m4));

    0.

let Responder(name: name, sk : skey) =
    event responderStarted(name);

    (* rcv m1 & send m2 *)
    in(c, m1: ke_m1);
    new rand: randomness;
```

```
    let m2 = ke2(rand, m1) in
    out(c, m2);
    let ctxt = (m1, m2) in
    let kek = keK2(rand, m1) in

    (* rcv m3 & send m4 *)
    in(c, (c0: bitstring, c1: bitstring));
    let m3 = (c0, c1) in
    let x = adec(sdec(c0, bitToKey(kek)), sk) in
    let k' = bitToKey(h((kek, x, ctxt))) in
    let decoded = sdec(c1, k') in
    let cert = p1(decoded) in
    let sig = p2(decoded) in
    let aPK = caCertKey(cert) in
    let aName = caCertName(cert) in
    let ctxt2 = (aName, name, m1, m2, m3) in
    let m4 = h((x, ctxt2)) in
    let caV = caCertValid(cert) in
    let sigV = vfySig((aName, name, c0, ctxt), sig, aPK) in
    let ctxt3 = (aName, name, m1, m2, m3, m4) in
    let k = h((kek, x, ctxt3)) in
    let ps = isPartyOk(aName) in

    new randomOutput : bitstring;
    new randomKey : bitstring;
    event responderDone(name, k, m1, m2, m3, m4, caV, sigV, ps)
   ;

    out(c, if (name <> psName(AB, psHONEST) || ps = psHONEST)
  && caV = caVALID && sigV = sigVALID then m4 else
  randomOutput);
    out(c, if (name <> psName(AB, psHONEST) || ps = psHONEST)
  && caV = caVALID && sigV = sigVALID then k else createK(m1,
  m2, m3, m4));

    0.

let SessionPair(name : name, pke : skey, sgn : sskey) =
    out(c, name);
    insert t(name, pk(pke), spk(sgn));
    Initiator(name, sgn) | Responder(name, pke).

let Party(name : name, pke : skey, sgn : sskey) =
```

```
    out(c, pk(pke));
    out(c, spk(sgn));

    if isPartyOk(name) = psCORRUPT then
        out(c, (pke, sgn, caCert(name, spk(sgn))));
        !SessionPair(name, pke, sgn)
    else
        !SessionPair(name, pke, sgn)
    .

let GenParty(ps : partyStatus) =
    new nOrig: name;
    new sk: skey;
    new ssk: sskey;
    let n = psName(nOrig, ps) in
    Party(n, sk, ssk).

process
    new A_pke: skey; new B_pke: skey; new A_sgn: sskey; new B_sgn: sskey;

    Party(psName(A, psHONEST), A_pke, A_sgn) | Party(psName(B, psHONEST), B_pke, B_sgn)
     | SessionPair(psName(AB, psHONEST), diff[A_pke, B_pke], diff[A_sgn, B_sgn])
        | (!GenParty(psHONEST)) | (!GenParty(psCORRUPT))
```

Listing 4.3: Proving s-MITM Privacy of Π_{Gen} (ProVerif)

Forward Privacy Formulation. In order to test forward privacy, we modified the previous code by adding the following command.

```
set attacker = passive.
```

This makes the adversary unable to create their own messages. Instead they can only read messages and do their own computations. Furthermore we now unconditionally reveal all information in the subprocess *Party*. For reference, we show the full code below.

```

(* ######## Settings ####### *)
set attacker = passive.

(* ######## Basics ######### *)
(* Symm Encr *)
```

```
type key.
fun senc(bitstring, key): bitstring.
fun sdec(bitstring, key): bitstring.
equation forall m: bitstring, k:key; sdec(senc(m,k),k) = m.
fun bitToKey(bitstring) : key [typeConverter].

(* Asymm Encr *)
type skey.
type pkey.
fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
fun adec(bitstring, skey): bitstring.
equation forall m: bitstring, k: skey; adec(aenc(m, pk(k)), k)
    = m.


(* Signatures *)
type sskey.
type spkey.
type sigValidity.
const sigVALID : sigValidity.
fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
fun vfySig(bitstring, bitstring, spkey) : sigValidity.
reduc forall m: bitstring, k: sskey; getmess(sign(m, k)) = m.
equation forall m: bitstring, k: sskey; vfySig(m, sign(m,k),
    spk(k)) = sigVALID.

(* Randomness *)
type randomness.

(* Hash *)
fun h(bitstring): bitstring.

(* Key Exchange *)
type ke_m1.
type ke_m2.
fun ke1(randomness) : ke_m1.
fun ke2(randomness, ke_m1) : ke_m2.
fun keK1(randomness, ke_m2) : bitstring.
fun keK2(randomness, ke_m1) : bitstring.
equation forall r1, r2 : randomness; keK1(r1, ke2(r2, ke1(r1)))
    = keK2(r2, ke1(r1)).
```

```
(* ####### General ######### *)
channel c.

type name. (* Identify parties *)
const A, B, AB : name.
table t(name, pkey, spkey). (* Table for retrieving pks *)

(* Certificates *)
type caCertValidity.
const caVALID: caCertValidity.
fun caCert(name, spkey) : bitstring [private].
fun caCertName(bitstring) : name.
fun caCertKey(bitstring) : spkey.
fun caCertValid(bitstring) : caCertValidity.
equation forall n: name, k: spkey; caCertName(caCert(n, k)) = n.
equation forall n: name, k: spkey; caCertKey(caCert(n, k)) = k.
equation forall n: name, k: spkey; caCertValid(caCert(n, k)) = caVALID.

(* Tuple Deconstructor Helper *)
fun p1(bitstring): bitstring.
fun p2(bitstring): bitstring.
equation forall x : bitstring, y : bitstring; p1((x, y)) = x.
equation forall x : bitstring, y : bitstring; p2((x, y)) = y.

(* Events *)
(* event initAccepts(k, m1, m2, m3, m4, expectedM4). *)
event initiatorDone(bitstring, ke_m1, ke_m2, bitstring, bitstring, bitstring).

(* event responderDone(k, m1, m2, m3, m4, caVal, vfySig). *)
event responderDone(bitstring, ke_m1, ke_m2, bitstring, bitstring, caCertValidity, sigValidity).

(* ####### Queries ######### *)
(* Sanity check. Can the protocol be completed successfully? *)
(* query k, m3, m4: bitstring, m1: ke_m1, m2: ke_m2; event(initiatorDone(k, m1, m2, m3, m4, m4)) && event(responderDone(k, m1, m2, m3, m4, caVALID, sigVALID)). *)

(* Explicit Entity Authentication *)
```

```
(* query k, m3, m4: bitstring, m1: ke_m1, m2: ke_m2; event(initiatorDone(k, m1, m2, m3, m4, m4)) ==> event(responderDone(k, m1, m2, m3, m4, caVALID, sigVALID)).   *)

(* ####### Protocol ######## *)
let Initiator(name: name, sgn : sskey) =
    in(c, peer : name); (* Receive adversary instruction of whom to contact. *)
    get t(=peer, bPKE : pkey, bSgn : spkey) in

    (* m1 *)
    new rand : randomness;
    let m1 = ke1(rand) in
    out(c, m1);

    (* rcv m2 & calc key *)
    in(c, m2 : ke_m2);
    let kek = bitToKey(keK1(rand, m2)) in

    (* send m3 *)
    new x : bitstring;
    let ctxt = (m1, m2) in
    let k' = bitToKey(h((kek, x, ctxt))) in
    let c0 = senc(aenc(x, bPKE), kek) in
    let c1 = senc((caCert(name, spk(sgn)), sign((name, peer, c0, ctxt), sgn)), k') in
    let m3 = (c0, c1) in
    out(c, m3);

    (* rcv m4 *)
    in(c, m4 : bitstring);
    let ctxt2 = (name, peer, m1, m2, m3) in
    let ctxt3 = (name, peer, m1, m2, m3, m4) in
    let expectedM4 = h((x, ctxt2)) in
    let k = h((kek, x, ctxt3)) in
    event initiatorDone(k, m1, m2, m3, m4, expectedM4);
    out(c, k);

    0.

let Responder(name: name, sk : skey) =
    (* rcv m1 & send m2 *)
    in(c, m1: ke_m1);
```

```
    new rand: randomness;
    let m2 = ke2(rand, m1) in
    out(c, m2);
    let ctxt = (m1, m2) in
    let kek = keK2(rand, m1) in

    (* rcv m3 & send m4 *)
    in(c, (c0: bitstring, c1: bitstring));
    let m3 = (c0, c1) in
    let x = adec(sdec(c0, bitToKey(kek)), sk) in
    let k' = bitToKey(h((kek, x, ctxt))) in
    let decoded = sdec(c1, k') in
    let cert = p1(decoded) in
    let sig = p2(decoded) in
    let aPK = caCertKey(cert) in
    let aName = caCertName(cert) in
    let ctxt2 = (aName, name, m1, m2, m3) in
    let m4 = h((x, ctxt2)) in
    let caV = caCertValid(cert) in
    let sigV = vfySig((aName, name, c0, ctxt), sig, aPK) in
    let ctxt3 = (aName, name, m1, m2, m3, m4) in
    let k = h((kek, x, ctxt3)) in

    new r : bitstring;

    event responderDone(k, m1, m2, m3, m4, caV, sigV);
    out(c, if caV = caVALID && sigV = sigVALID then m4 else r);
    out(c, k);

    0.

let SessionPair(name : name, pke : skey, sgn : sskey) =
    insert t(name, pk(pke), spk(sgn));
    Initiator(name, sgn) | Responder(name, pke).

let Party(name : name, pke : skey, sgn : sskey) =
    out(c, name);
    out(c, pk(pke));
    out(c, spk(sgn));

    out(c, (pke, sgn, caCert(name, spk(sgn)))); (* leak *)

    !SessionPair(name, pke, sgn).
```

```
process
    new A_pke: skey; new B_pke: skey; new A_sgn: sskey; new
   B_sgn: sskey;

    Party(A, A_pke, A_sgn) | Party(B, B_pke, B_sgn) |
   SessionPair(AB, diff[A_pke, B_pke], diff[A_sgn, B_sgn])
        | !(new n: name; new sk: skey; new ssk: sskey; Party(n,
    sk, ssk))
```

Listing 4.4: Proving Forward Privacy of Π_{Gen} (ProVerif)

4.4 Results

In this chapter we discussed the encoding of Π_{Gen} both for the Tamarin Prover and ProVerif. While our Tamarin encoding works for proving simple lemmas, proving its observational equivalence (for strong MITM privacy) turns out to be infeasible. We argued why switching to ProVerif is beneficial both for increasing the confidence in the result (due to a more natural encoding) as well the efficiency of the computation. Indeed ProVerif successfully proved the strong MITM privacy and forward privacy of Π_{Gen}.

CHAPTER 5

Conclusion

Recently there has been an increasing interest in privacy guarantees of protocols, leading to the work of Zhao [Zha16] and Schäge, Schwenk and Lauer [SSL20]. As we showed, and also as noted by the authors, the privacy of the protocols shown in both of these works are vulnerable to MITM attacks, that do not complete the respective session. We presented a novel model and several protocols that account for these types of attacks. Unlike the protocols in the previously mentioned literature, most of our protocols require the initiator to know the intended recipient's public key (or obtain it through some safe method). This seems to be a reasonable requirement for most application scenarios, but even if this cannot be achieved, our protocol Π_{ss} can be used. Π_{ss} reaches the same privacy properties as the protocols from the previously mentioned literature in our model, as well as providing additional defense against MITM attackers. In order to achieve this, Π_{ss} uses a shared secret that needs to be distributed to all honest users, e.g. when the CA signs their public keys. This shared secret is only needed for our notion of weak MITM privacy. Leaking this secret does not affect security (key indistinguishability) and the recently studied privacy notion in the literature, which we call completed session privacy, still holds.

Furthermore the model we presented is modular and allows to classify protocols based on the classical security notion of key indistinguishability as well as multiple, novel privacy notions (weak MITM privacy, strong MITM privacy, forward privacy).

We showed several protocols that require 2, 3 or 4 moves and fulfill varying degrees of privacy, with the 4-move protocol Π_{Gen} fulfilling all of them. This result was proven classically, as well as using automated verification tools to give additional confidence. We evaluated the Tamarin Prover as well as ProVerif. Our Tamarin formulation could not be processed without running out of memory on a machine with around 60 GB available RAM. We argued why switching to ProVerif is beneficial both for increasing the confidence in the result (due to a more natural encoding, in particular of if/then/else-statements) as well improving the efficiency of the computation. Indeed, the ProVerif formulation was proven successfully.

As a potential direction for future research, one might investigate the possibility of a one-move PPAKE. This could potentially be realized by using puncturable encryption, but there are some hurdles that need to be overcome. Zhao[Zha16] presents the protocol Higncryption which can be transformed into a one-move AKE that provides initiator privacy. However, it would not provide key indistinguishability or forward privacy in our model, since we allow the corruption of the responder.

List of Figures

List of Tables

Index

Bibliography

[ABB+04] William Aiello, Steven M. Bellovin, Matt Blaze, Ran Canetti, John Ioannidis, Angelos D. Keromytis, and Omer Reingold. Just fast keying: Key agreement in a hostile internet. *ACM Trans. Inf. Syst. Secur.*, 7(2):242–273, 2004.

[ABF+19] Ghada Arfaoui, Xavier Bultel, Pierre-Alain Fouque, Adina Nedelcu, and Cristina Onete. The privacy of the tls 1.3 protocol. *Proceedings on Privacy Enhancing Technologies*, 2019(4):190–210, 2019.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, Heidelberg, December 2001.

[BBM00] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274. Springer, Heidelberg, May 2000.

[BBS] Vincent Cheval Bruno Blanchet, Ben Smyth and Marc Sylvestre. Proverif 2.02pl1: Automatic cryptographic protocol verifier, user manual and tutorial. `https://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf`. Accessed: April 6th, 2021.

[BCF+13] Colin Boyd, Cas Cremers, Michele Feltz, Kenneth G. Paterson, Bertram Poettering, and Douglas Stebila. ASICS: Authenticated key exchange security incorporating certification systems. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 381–399. Springer, Heidelberg, September 2013.

[BFGJ17] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 651–681. Springer, Heidelberg, August 2017.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi

Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, pages 62–73. ACM, 1993.

[CCG+19] Katriel Cohn-Gordon, Cas Cremers, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. Highly efficient key exchange protocols with optimal tightness. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 767–797. Springer, Heidelberg, August 2019.

[CK02] Ran Canetti and Hugo Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 143–161. Springer, Heidelberg, August 2002. `http://eprint.iacr.org/2002/120/`.

[DGJ+21] David Derler, Kai Gellert, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-rtt key exchange. *J. Cryptol.*, 34(2):13, 2021.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[FH98] Paul Ferguson and Geoff Huston. What is a vpn? `https://www.semanticscholar.org/paper/What-Is-a-VPN-%E2%80%94-Part-I-Ferguson-Huston/1dbed25c3b13565073803c32489303e0094b020c`, 1998. Accessed 17-05-2021.

[GM15] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 305–320. IEEE Computer Society, 2015.

[KHN+14] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet key exchange protocol version 2 (ikev2). *RFC*, 7296:1–142, 2014.

[Kra03] Hugo Krawczyk. Sigma: The 'sign-and-mac' approach to authenticated diffie-hellman and its use in the ike protocols. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 400–425, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[LS17] Yong Li and Sven Schäge. No-match attacks and robust partnering definitions: Defining trivial attacks for security protocols is not trivial. In Bhavani M.

Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1343–1360. ACM Press, October / November 2017.

[OP01] Tatsuaki Okamoto and David Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In *International workshop on public key cryptography*, pages 104–118. Springer, 2001.

[RSW21] Sebastian Ramacher, Daniel Slamanig, and Andreas Weninger. Privacy-preserving authenticated key exchange: Stronger privacy and generic constructions. In *26th European Symposium on Research in Computer Security (ESORICS 2021)*. Springer, 2021. to be published.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.*, 2004:332, 2004.

[SSL20] Sven Schäge, Jörg Schwenk, and Sebastian Lauer. Privacy-preserving authenticated key exchange and the case of IKEv2. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 567–596. Springer, Heidelberg, May 2020.

[Tea] The Tamarin Team. Tamarin-prover manual. `https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf`. Accessed: April 6th, 2021.

[Zha16] Yunlei Zhao. Identity-concealed authenticated encryption and key exchange. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1464–1479, 2016.