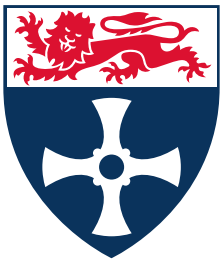
Machine-Checked Formalisation and Verification of Cryptographic Protocols



Roberto Metere

Supervisor: Dr. Changyu Dong

School of Computing
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

August 2020

To dad.
I will always hear his words that have been echoing in my mind along this journey.

> “Non sostenere mai niente che non possa anche dimostrare.”
>
> *Antonio Metere*

(Do never support anything you cannot prove.)

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Roberto Metere
August 2020

Acknowledgements

First and foremost, I would like to express my gratitude to my advisor Changyu Dong, without whom this research would not have been possible. He's just the best, he constantly mentored me during the ups and downs of this journey: he encouraged me when I was demoralised and always acknowledged my (small) successes. Thanks for his invaluable support toward shaping the kind of researcher I wanted to become. I would like to thank Feng Hao, who has been my second advisor before moving to Warwick; he has valued my work and helped my research in so many aspects with his experienced suggestions and collaboration. I am grateful to my colleague and friend Luca Arnaboldi who listened to (and have been annoyed by) my formal method ideas where few others would. I also thank all my colleagues I have met in the SRS group, Aad, Maryam and Ehsan, Siamak, Mario, Paulius, John, Peter, Sean, Artur, Uchechi, and all the others, for the professional and, more importantly, the recreational support, it is a great pleasure to be part of such an amazing (family) group. A special thanks goes to my current advisor Myriam Neaimeh at The Alan Turing Institute, London, and co-advisor Charles Morisset at Newcastle University, as they allowed me to organise my time always considering this goal.

Outside of Newcastle University, I am firstly grateful to Massimo Bartoletti at the University of Cagliari and to Roberto Guanciale at KTH Royal Institute of Technology, who let me appreciate the importance of formal methods in the reasoning for verification, which is somehow the virtual heart of this dissertation. I would like to express my eternal gratitude to Benjamin Grégoire for inviting me to visit the Inria research institute in France. He is a rare brilliant mind, whose ideas, along with a brief discussion with Gilles Barthe, have been of great help to the soundness of the last part of this dissertation. I also would like to thank Alley Stoughton at Boston University, who helped me very much at the beginning of my journey.

Outside of the research community, I would like to thank particularly Roberta Lecca, who suffered the countless week-ends I spent in the office. A special thanks to my family who always supported and encouraged me to pursue my dreams. I feel especially grateful to all my friends who have always let me feel close and welcome despite of the distance.

Abstract

Aiming for strong security assurance, researchers in academia and industry focus their interest on formal verification of cryptographic constructions. Automatising formal verification has proved itself to be a very difficult task, where the main challenge is to support generic constructions and theorems, and to carry out the mathematical proofs.

This work focuses on machine-checked formalisation and automatic verification of cryptographic protocols. One aspect we covered is the novel support for generic schemes and real-world constructions among old and novel protocols: key exchange schemes (Simple Password Exponential Key Exchange, SPEKE), commitment schemes (with the popular Pedersen scheme), sigma protocols (with the Schnorr's zero-knowledge proof of knowledge protocol), and searchable encryption protocols (Sophos).

We also investigated aspects related to the reasoning of simulation based proofs, where indistinguishability of two different algorithms by any adversary is the crucial point to prove privacy-related properties. We embedded information-flow techniques into the EasyCrypt core language, then we show that our effort not only makes some proofs easier and (sometimes) fewer, but is also more powerful than other existing techniques in particular situations.

Table of contents

Chapter 1

Introduction

1.1 Motivation

The high and increasing volume of communication exchanged through insecure channels, e.g. the Internet, confers increasing significance on security guarantees of cryptographic protocols. A cryptographic protocol is a set of algorithms performing a security-related function that employ cryptographic methods, e.g. composition of cryptographic primitives. The increasing complexity in the design of such protocols leads to more complex and longer proofs. For this reason, lack of rigour in the published research of the past few decades has been appreciated and debated [38, 93].

> "In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor."
>
> [Bellare and Rogaway (2004)]

> "Do we have a problem with cryptographic proofs? Yes, we do. The problem is that as a community, we generate more proofs than we carefully verify (and as a consequence some of our published proofs are incorrect)."
>
> [Halevi (2005)]

To address this problem, many tools have been developed based on different approaches [128, 120, 123, 129, 73, 74, 91, 64, 49, 127, 150, 90, 15, 29, 4, 80, 160, 51, 70, 12, 125, 45, 26, 20, 138, 69, 31] and all employ formal methods to mechanise the mathematical reasoning required to capture security properties; tools and approaches will be discussed in Section 2.6. The research community witnessed the effectiveness of these tools for verifying security properties and finding attacks. In a nutshell, formal models allow for mechanising proofs of security of cryptographic protocols with

respect to mathematical models. The benefits brought by formal methods are manifold: not only they improve rigour in cryptographic proofs, by also they offer further important and desirable properties as repeatability, reproducible results, automation on verification, and a more reliable design as a basis for real implementations [24]. The research questions in this field are therefore related to investigating the potential of the formal models used in cryptography by looking from (at least) two points of view: first, what security properties can be studied and what in details the currently available tools allow for reasoning about; and second, the degree of automation and support that can be obtained by using those tools.

1.1.1 The research problem

Novel protocols often base their construction on pre-existing cryptographic primitives and security properties used as building blocks. A modern cryptographer wanting to mechanise a formal proof is supposed to focus on the of the protocol she is designing. However, she will likely encounter an obstacle to overcome: not all cryptographic primitives and not all security properties are supported by those tools, even less by a single tool. The reason of such incompleteness is that not only *proofs* are generally difficult to mechanise, but also *modelling protocols* in a way that the model covers all the security aspects to analyse is difficult as well. So the cryptographer is left with two choices: she can either mechanise all the unsupported blocks by herself or leave a gap in the proof. The former choice would require to implement the unsupported blocks and prove their security; in many cases, it also requires to modify the source code of the tool. Needless to say, this would decouple the time that was originally dedicated to the protocol analysis. The latter choice would definitely leave part of the proof as *future works*, and some researchers might not like leaving such gaps, so that they might just give up using formal tools; also, sometimes it is not easy to model only part of the protocol and get satisfactory results by the partial mechanisation.

In summary, the problems are that the corpus of mechanised security analysis of cryptographic primitives and cryptographic protocols is not complete, and that proofs structures still require much improvement to tame their complexity. Those problems come from the generally difficult and time-consuming tasks of **modelling protocols and security properties**, and **proving security claims**. Clearly, solving those problems would require an immense effort, so we could not pretend to solve them, but we contributed toward their solution: i) by re-modelling protocols to analyse more security properties than previous models; ii) by adding support to new cryptographic primitives in existing tools or verification environments; iii) by enriching the growing

corpus of mechanised proofs of novel constructions; and iv) by implementing a proof technique to ease some special (but common) proof structures.

1.2 Contributions

With the strong motivation discussed in the previous Section 1.1, we planned our work to touch different aspects of the formal verification process in different cryptographic models that can capture different security aspects and properties.

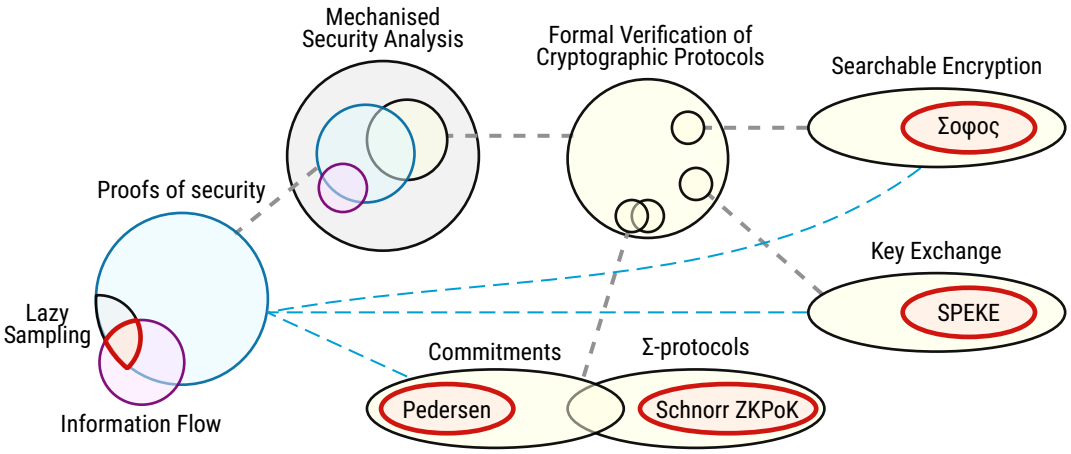


Fig. 1.1 A scheme of our contributions (thick red ellipses in the diagram) in the context of mechanised security analysis.

Referring to Figure 1.1, we tackled different problems in the context of mechanised security analysis, and our contribution can be summarised with the following list.

- Our first contribution came with the opportunity to formally specify the Simple Password Exponential Key Exchange (SPEKE) protocol, which is part of the standard ISO/IEC 11770-4. Our formalisation covers from its first specification [105] to our last (now included in the standard) which came along with their formal verification of several attacks and security properties [95, 104]. The SPEKE protocol is a very lightweight protocol for letting machines exchange a key in the setting where they already share a (weak) password. We completed our implementation in a formalism referred as the symbolic model which will be explained in Section 2.3.1. With this work, we gained experience in the formal tool ProVerif [49] which we used to formalise a list of security properties and attacks, including malleability. Importantly, the SPEKE protocol

has been verified in the past in the AVISPA tool [160], but their model was not capturing all the aspects as we do and could *not* find any attacks to the protocol.

- Our second contribution was providing generic construction of commitment protocols and sigma protocols that can be instantiated with any implementations. Commitment schemes are a cryptographic primitive that have been widely used by its own or as a building block in other protocols; for example, verifiable secret sharing, zero-knowledge proofs, and e-voting [131, 87, 146]. Before our work, one would have had to manually write all the structures, cryptographic experiments and theorems for each implementation of the above mentioned protocols. As motivating example, we successfully used our implementation to verify the Pedersen commitment scheme [136, 126] and the Schnorr protocol of zero knowledge proof of knowledge (as a sigma protocol), see Chapter 4. This work has been carried on in a formalism denoted as the computational model which will be explained in Section 2.3.2.

- To study more complex protocols, we further investigated searchable encryption protocols. We found those very challenging from the point of view of automation, as no proofs were available (in tools) in searchable encryption that could cope against adaptive adversaries. Our contribution toward this direction starts including a precise formalism for index based symmetric searchable encryption protocols. As a motivating example, we proved the forward secrecy of the Sophos scheme [52], we discuss it more in details in Chapter 6. During this proof that took more than one year, we faced several challenges, including new theories and new theorems extending the already existing theories in the formal language we chose, EasyCrypt.

- Furthermore for the proof of Sophos, we introduced some information flow extending the core logic of EasyCrypt, see Chapter 5. The aim of our extension is to simplify indistinguishability proofs involving the theoretical strategy of lazy sampling [39], whose pattern is common in security theorems involving oracles. We relate to the current implementation in EasyCrypt of lazy sampling, and we propose a novel strategy based on the labelling of variables typical of information flow analysis. This was very challenging, as we needed to modify the source code of EasyCrypt; however, we succeeded in our intent and many other proofs can benefit by our strategy.

1.3 Structure

In Chapter 2, we introduce some concepts that are required to follow smoothly the content of the next chapters. The first part of the chapter illustrates formal definitions of security and cryptographic games, on which the properties we discuss in the dissertation are based. The last part of the chapter discusses cryptographic models to design protocol and that are the basis for their security analysis.

We illustrate our work in details grouped by two main topics: the symbolic model, Chapter 3, and the computational model, Chapters 4, 6, and 5. In those chapters, we first highlight the motivation and the state-of-the-art upon which we based our contribution. Then, we will show in details our contribution and draw conclusions.

Finally in Chapter 7, we will summarise conclusions of the whole dissertation and discuss limitations, points of improvement and future works.

1.4 Publications

Here is a list of co-authored publications related to security, cryptography and formal verification; the publications relevant to this thesis are put first in the following list and marked with a gray line.

Analyzing and patching SPEKE in ISO/IEC – *Chapter 3*
Feng Hao, Roberto Metere, Siamak F. Shahandashti, Changyu Dong
TIFS 2018 (IEEE Transactions on Information Forensics and Security)

Automated Cryptographic Analysis of the Pedersen Commitment Scheme – *Chapter 4*
Roberto Metere, Changyu Dong
MMM-ACNS 2017 (International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security)

End-to-End Verifiable E-Voting Trial for Polling Station Voting at Gateshead
Feng Hao, Shen Wang, Samiran Bag, Rob Procter, Siamak F Shahandashti, Maryam Mehrnezhad, Ehsan Toreini, Roberto Metere, Lana Liu
IEEE Security & Privacy, 2020

Modelling Load-Changing Attacks in Cyber-Physical Systems
Luca Arnaboldi, Ricardo M Czekster, Roberto Metere, Charles Morisset
Electronic Notes in Theoretical Computer Science, 2020

Poster: Towards a Data Centric Approach for the Design and Verification of Cryptographic Protocols
Luca Arnaboldi, Roberto Metere
CCS 2019, (Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security)

TrABin: Trustworthy analyses of binaries
Andreas Lindner, Roberto Guanciale, Roberto Metere
Science of Computer Programming, 2018

Incentive-driven attacker for corrupting two-party protocols
Yilei Wang, Roberto Metere, Huiyu Zhou, Guanghai Cui, Tao Li
Soft Computing, 2018

Socially-conforming cooperative computation in cloud networks
Tao Li, Brij Bhooshan Gupta, Roberto Metere
Journal of Parallel and Distributed Computing, 2018

Sound transpilation from binary to machine-independent code
Roberto Metere, Andreas Lindner, Roberto Guanciale
Brazilian Symposium on Formal Methods, 2017

A certificateless signature scheme and a certificateless public auditing scheme with authority trust level 3+
Fei Li, Dongqing Xie, Wei Gao, Kefei Chen, Guilin Wang, Roberto Metere
Journal of Ambient Intelligence and Humanized Computing, 2017

Efficient delegated private set intersection on outsourced private datasets
Aydin Abadi, Sotirios Terzis, Roberto Metere, Changyu Dong
IEEE Transactions on Dependable and Secure Computing, 2017

Chapter 2

Background

The security protocols discussed in this dissertation rely on common cryptographic notions and definitions. This Section introduces such preliminaries and cryptographic models used to reason about security of protocols.

2.1 Security notions in cryptography

In the past, cryptographers struggled to invent cryptographic systems that could provide information-theoretic security that could be applicable to diverse contexts, such as secret communication, integrity or availability. Information-theoretic security schemes are unbreakable by construction, even by an adversary with unbounded computational power. For example, a well known information-theoretic secure scheme is the one-time-pad. Unfortunately, it comes at the price of secretly pre-sharing a key of the same length of the message and can be used only once; so the communicating parties would need as many pre-shared keys as the messages they intend to exchange in the future. This is not just a problem of one-time-pad, but a problem of all information-theoretic secure schemes [147].

To circumvent this problem, a relaxed definition of security called *computational security*, can be adopted. Computational security limits to adversaries with polynomially bounded computational resources, and with a *small* probability of breaking the security. This notion allows for using a small key to encrypt a large amount of plaintexts; it moreover allows for asymmetric encryption. Computational security relies on the truthfulness of the famous open-problem in complexity theory $\mathcal{P} \neq \mathcal{NP}$ [82], whose discussion is out of our scope.

2.1.1 Negligible functions

Computational security relaxes standard security properties definitions by letting them fail at most for a negligible probability. Negligible functions are the necessary formality to capture such a concept. The family of *negligible functions* describes those functions that decrease faster than the inverse to a polynomial.

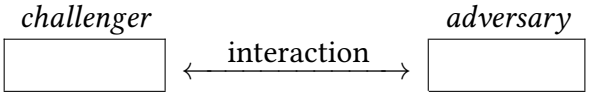
Definition 2.1.1 (Negligible function)**.** *A function* $\mu : \mathbb{N} \to \mathbb{R}$ *is* negligible *if, for all positive natural* c*, there exists a natural number* n_0 *such that*

$$\forall n \in \mathbb{N},\ n_0 < n \Rightarrow |\mu(n)| < \frac{1}{n^c}.$$

Informally, if n is a security parameter and a security property holds up to negligible probability on n, then we can *tune* the strength of a security property by increasing or decreasing n. Increasing the security parameter n increases both the time required by honest parties to run the protocol and the breaking algorithm of the attackers. The key concept is that while the former increases polynomially (as it is an efficient algorithm), the latter increases super-polynomially, e.g. exponentially.

2.1.2 Cryptographic experiments

In cryptography, security properties are often related to cryptographic experiments, or **games**, where a benign entity called the *challenger* plays against the adversary. Both the challenger and the adversary are interacting probabilistic processes that



allow the game to be modelled as a probability space. Finally, a security property is captured as a statement over the probability of an event E related to a cryptographic experiment that equal or are negligibly close to a *target probability* p:

$$\exists \mu,\ \Pr[E] \leq p + \mu(n)$$

where μ is a negligible function over the security parameter n. For example, an event E can be the adversary winning the game, and the security property may require that the probability of such an event must be negligible (or negligibly close to 0).

2.1.3 Computational indistinguishability

Computational indistinguishability is a central notion to the theory of cryptography. We provide a definition based on cryptographic game, where the challenger plays against an adversary, called a distinguisher, who is challenged to tell two probability distributions apart. For example, the security of an encryption system is defined as indistinguishability of ciphertexts, where the adversary is asked for two plaintexts and then challenged with an encryption of one of them; the adversary should not be able to guess which plaintext corresponds to the challenge better than a coin toss.

And indistinguishability experiment should capture the security of the protocol for the security parameter n tending to infinite. The necessary formalism to have this asymptotic approach is given by an infinite sequences of random bitstrings, called *probabilistic ensembles*, that can be efficiently sampled. The following formal notation is adapted from of Hazay and Lindell [99], Goldreich [85], and Katz and Lindell [109].

Notation 2.1.1 (Probability ensemble)**.** *Given an index value $a \in \{0,1\}^\star$, and a security parameter $n \in \mathbb{N}$, we denote a* probability ensemble *$\mathcal{X}$ indexed by a and n the set of all random variables $X_{a,n}$ for all a and all n:*

$$\mathcal{X} = \{X_{a,n}\}_{a\in\{0,1\}^\star, n\in\mathbb{N}}.$$

Additionally, there exists an efficient sampling algorithm $\mathcal{S}$ such that for all a and n, $\mathcal{S}(1^n, a)$ and $X_{a,n}$ are identically distributed.

In the notation and throughout the whole dissertation, we denote as $\{0,1\}^\star$ the set of bit strings of any length, and as 1^n the n-bit string of ones: $11\ldots1$ (length n) and is the standard convention in cryptography that emphasises that the length is provided or known to the algorithm, in some security definitions n is called the *security parameter*.

The input of the parties in a protocol or in a cryptographic experiment is described by the value a indexing a probability ensemble $\mathcal{X}$ [99], while the description of the output is included in the random variable $X_{a,n} \in \mathcal{X}$. The random variable $X_{a,n} \in \mathcal{X}$ is what the adversary will finally inspect.

We are interested in the indistinguishability between two different constructions Π_0 and Π_1 of the same protocol, i.e. they provide the same functionality. Their output can be seen as two random variables $X_{a,n}$ and $Y_{a,n}$ in the ensembles $\mathcal{X}$ and $\mathcal{Y}$ respectively to the games. The result of the adversarial strategy, or any other event capturing security, can be seen as the event E as a function over one such variable. So, we can relate Π_0 and Π_1 through the probability of events E_0 and E_1, whose (absolute) difference is at the basis of the concepts of indistinguishability:

$$\left|\mathsf{Pr}\left[E_X\left(X_{a,n}\right)\right] - \mathsf{Pr}\left[E_Y\left(Y_{a,n}\right)\right]\right|,$$

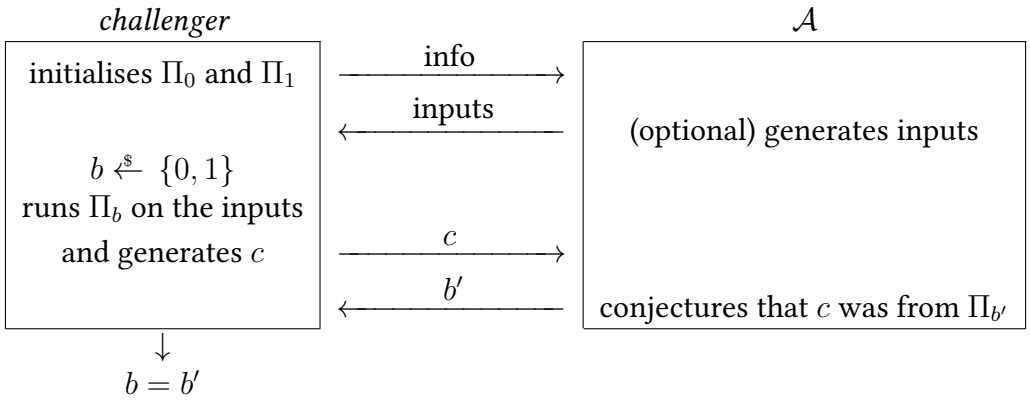
where $X_{a,n}$ and $Y_{a,n}$ are random variable in the ensembles $\mathcal{X}$ and $\mathcal{Y}$. The event we generally refer to in cryptographic lemmas relate to the output of an adversary to be able to determine a predicate over the *challenging* string. Finally the *event* is the experiment outputting 1 whenever the adversary guesses correctly. So we write $\mathsf{Pr}\left[\mathcal{A}\left(X_{a,n}\right) = y\right]$ for the probability of the event when the adversary provided with $X_{a,n}$ and outputs y. We equivalently write $\mathsf{Pr}\left[\mathcal{A}\left(\Pi\right) = y\right]$ for $\mathsf{Pr}\left[\mathcal{A}\left(X_{a,n}\right) = y\right]$ if $X_{a,n}$ relates to the construction Π.

The ability of the adversary, or distinguisher, to guess is measured by her probability to win a distinguishing game, which is called the *advantage* of the adversary. In the game, the adversary is provided a trace of either Π_0 or Π_1 and outputs $b \in \{0,1\}$ if she thinks that the trace corresponds to Π_b. More precisely, an indistinguishability game is a cryptographic experiment $\mathsf{Exp}_{\Pi_0,\Pi_1}$ that relates the two constructions Π_0 and Π_1 as illustrated in 2.1 and described as follows:

1. The challenger initialises the protocols in Π_0 and Π_1 and provides some related parameters to the distinguisher.

2. Depending on the security property and the adversarial capabilities, the distinguisher may be allowed to decide (some of the) inputs of the parties.

Fig. 2.1 Flow of the experiment for indistinguishability two protocols described by two constructions Π_0 and Π_1.



3. The challenger flips a coin to generate the challenge using either Π_0 or Π_1, then he sends the challenge to the adversary.

4. The adversary accepts the challenge and answers back with 0 if associates the challenge to Π_0, 1 otherwise.

5. Finally, the challenger outputs whether the adversary guesses correctly or not.

We notice that the indistinguishability experiment here defined is a trivial generalisation of common indistinguishability experiments [148, 109].

Definition 2.1.2 (Advantage)**.** *Given an experiment* $\mathsf{Exp}^{\mathcal{A}}_{\Pi_0,\Pi_1}$ *of indistinguishability over two constructions* Π_0 *and* Π_1 *of the same functionality run against the adversary* $\mathcal{A}$*, the distinguishing advantage* $\mathbf{Adv}$ *of the adversary is defined as*

$$\mathbf{Adv}^{\mathsf{Exp}}_{\Pi_0,\Pi_1}(1^n) \stackrel{\text{def}}{=} \left|\mathsf{Pr}\left[\mathcal{A}(\Pi_0) = 1\right] - \mathsf{Pr}\left[\mathcal{A}(\Pi_1) = 1\right]\right|.$$

The events $[\mathcal{A}(\Pi_b) = b']$ happen when the adversary is provided with the challenge of Π_b and outputs b', i.e. guesses correctly in the case $\Pi_{b'}$ generated the challenge, incorrectly in the case $\Pi_{1-b'}$.

The advantage allows to easily define the computational indistinguishability if related to a negligible function as upper bound. Our definition of computational indistinguishability is based on the definition of *computational indistinguishability of ensembles* by Katz and Lindell [109], with the only change that the ensembles are generated by an indistinguishability experiment.

Definition 2.1.3 (Computational indistinguishability)**.** *Given an indistinguishability experiment* $\mathsf{Exp}^{\mathcal{A}}_{\Pi_0,\Pi_1}$ *over two constructions* Π_0 *and* Π_1 *of the same protocol run against the adversary* $\mathcal{A}$*, the probabilistic ensembles* $\mathcal{X}$*, due to* Π_0*, and* $\mathcal{Y}$*, due to* Π_1*, are* computationally indistinguishable*, denoted as* $\mathcal{X} \stackrel{c}{\equiv} \mathcal{Y}$*, if and only if for any probabilistic polynomial-time adversary* $\mathcal{A}$ *exists a negligible function* μ *such that*

$$\mathbf{Adv}^{\mathsf{Exp}}_{\Pi_0,\Pi_1}(1^n) \leq \mu(n)$$

where n is the security parameter related to the length of protocols' secrets.

For simplicity, the constructions themselves are said to be indistinguishable, without explicitly referring to their probabilistic ensembles. In such cases, given two constructions Π_0 and Π_1, we would write

$$\Pi_0 \overset{c}{\equiv} \Pi_1.$$

2.2 Proof structures in cryptography

Modern cryptography relaxes the concept of security to *computational security*, the adversary is seen as a modern computer, whose resources are bounded to be polynomial on a security parameter. In this scenario, the probability to break security is required to be *negligible*, and it is related to the computational hardness of the adversarial strategy. In our work, we focus on security properties defined by mathematical theorems, that can be therefore *provable*, and in particular we discuss game-based proofs that can be automatised in a tool run by a machine. The structure of proofs in computational security vary, but often are game-based, that is captured by a cryptographic experiment again an efficient adversary, as introduced in Section 2.1.2. In this section we will briefly discuss the concepts of reduction, sequence of games, simulation-based proofs, and proofs where the adversary is provided with oracle[1] access to functionalities. All those concepts are not disjoint: for example, both game-based and simulation-based proofs can be structured as a sequence of games.

2.2.1 Reduction

In cryptography, some security properties are defined on hardness assumptions. For example in the Diffie-Hellman key exchange protocol, confidentiality of the key is based on the hardness of computing the discrete logarithm. The proof of similar properties is usually carried out by showing how to transform the original problem into an instance of the hard problem. This is a common approach in complexity theory called *reduction*. For example, the security of the RSA cryptosystem can be reduced to the factorisation of integers, which is known to be hard. In cryptography, proofs by reduction are often argued by contradiction: we suppose that the original problem is easy to solve, then we show that we can break the hardness of some other problem known (or assumed) to be hard. This contradiction implies that the original problem is hard too. For example, we assume that RSA is not secure, then we show that this implies that integer factorisation can be solved efficiently.

2.2.2 Sequence of games

The proof of security properties based on games can become complicated, long and very difficult to follow, thus to verify. To tame such complexity, the original game is

[1] In cryptography, oracles are black-box efficient abstractions of functionalities that can be used in experiments to capture certain security properties.

transformed to another game whose security is known, so the proof becomes proving the soundness of the transformation. More in details, a transition would transform the probability over an event E related to a game G to the probability over the event $\widetilde{E}$ related to another game $\widetilde{G}$. Assume we have to prove that

$$\exists \mu,\ \mathsf{Pr}\left[E\right] \leq p + \mu\left(n\right),$$

where μ is a negligible function over the security parameter n and p is the target probability that capture security. If we can prove that the difference between $\mathsf{Pr}\left[E\right]$ and $\mathsf{Pr}\left[\widetilde{E}\right]$ is negligible, and if we can also prove that

$$\exists \mu,\ \mathsf{Pr}\left[\widetilde{E}\right] \leq p + \mu\left(n\right),$$

then the original statement is easily provable. If the above is a known result, e.g. hardness of factoring integers, the only piece to prove is the equivalence of G and $\widetilde{G}$ up to a negligible probability. We would write the transition as $G \rightsquigarrow \widetilde{G}$.

When a transition from a game G to a game $\widetilde{G}$ is still too complex, then multiple intermediate games $G_1, G_2, \ldots, G_N$ are created, and the proof follows an approach called *sequence-of-games* [148, 93, 40]. A proof structured as a sequence of $N + 2$ games requires to prove $N + 1$ sub-goals, each of which needs to show that the advantage in indistinguishability between all adjacent games (often called the *distance*) is negligible.

$$G \rightsquigarrow G_1 \rightsquigarrow G_2 \rightsquigarrow \cdots \rightsquigarrow G_N \rightsquigarrow \widetilde{G}$$

By transitivity, one finally has $G \rightsquigarrow \widetilde{G}$, and the benefit of this structure can be appreciated when each sub-goal is verifiable with little effort.

Generally, the transition from a game G_i to the next G_{i+1} is of three types: it can be based on *indistinguishability*, it can be based on *failure events*, or it can be a *bridging step* that we discuss below. There is no recipe on how to create such games, and it is matter of the taste and creativity of the cryptographer carrying out the proof.

Indistinguishability

In a transition based on indistinguishability, a game G_{i+1} is created from G_i with a small change that, if detected by the adversary, it implies the existence of an efficient algorithm to tell the two games apart. If we relate two events E_i defined on G_i and E_{i+1} defined on G_{i+1} to two probability ensembles, then we can refer to the computational indistinguishability of Definition 2.1.3. In brief, the indistinguishability to prove requires to show that $|\mathsf{Pr}\left[E_i\right] - \mathsf{Pr}\left[E_{i+1}\right]|$ is negligible.

Failure events

Assume we have two games G_i and G_{i+1} that proceed identically unless the event F (for failure) happens. Events such as F are often called *bad events* and need to be negligible, e.g. a repeated value sampled from a uniformly random distribution. The difference in probability of two events E_i defined on G_i and E_{i+1} defined on G_{i+1} can be bound by the probability of the occurrence of the failure event F in

either game [27]. Formally, the bound is provided by Lemma 2.2.1, called *fundamental lemma* [40, 27] or *difference lemma* [148]. We adapted the lemma by Barthe et al. [27] to our notation so far, as it is the most appropriate to code-based games, where termination is also explicitly considered.

Lemma 2.2.1 (Difference lemma). *Let G_i and G_{i+1} be two games, E_i an event defined on G_i, E_{i+1} an event defined on G_{i+1} and F an event defined in both games. If both G_i and G_{i+1} terminate, then*

$$\mathsf{Pr}\left[E_i \wedge \neg F\right] = \mathsf{Pr}\left[E_{i+1} \wedge \neg F\right] \Rightarrow \left|\mathsf{Pr}\left[E_i\right] - \mathsf{Pr}\left[E_{i+1}\right]\right| \leq \mathsf{Pr}\left[F\right].$$

This is a simple calculation.

$$\begin{aligned}\left|\mathsf{Pr}\left[E_i\right] - \mathsf{Pr}\left[E_{i+1}\right]\right| &= \left|\mathsf{Pr}\left[E_i \wedge F\right] + \mathsf{Pr}\left[E_i \wedge \neg F\right] - \mathsf{Pr}\left[E_{i+1} \wedge F\right] - \mathsf{Pr}\left[E_{i+1} \wedge \neg F\right]\right| \\ &= \left|\mathsf{Pr}\left[E_i \wedge F\right] - \mathsf{Pr}\left[E_{i+1} \wedge F\right]\right| \\ &\leq \mathsf{Pr}\left[F\right]\end{aligned}$$

The second equality follows from the assumption $\mathsf{Pr}\left[E_i \wedge \neg F\right] = \mathsf{Pr}\left[E_{i+1} \wedge \neg F\right]$. The inequality follows by the fact that both $\mathsf{Pr}\left[E_i \wedge F\right]$ and $\mathsf{Pr}\left[E_{i+1} \wedge F\right]$ are bounded by $\mathsf{Pr}\left[F\right]$.

In code-based proofs, the failure condition can be implemented by setting a flag variable to `true` when a condition capturing the bad event holds. For example, the condition may test if a collision is found in a random function, in which case the flag is set to `true`.

Bridging steps

It is also common to see transitions to games whose changes are purely conceptual and transform a game to another equivalent game, e.g. maps reimplemented as lists or statements swapped with others: those are commonly called *bridging steps* and they prepare the ground for a transition of one of the above two types. While in principle, bridging steps may seem unnecessary, without it, the proof would be much harder to follow [148].

2.2.3 Simulation-based proofs

Simulation-based proofs are based on the concept of computational indistinguishability. We define some of our security properties, for the most for searchable encryption in Chapter 6, using the standard simulation model of secure computation [85], where the adversarial entity is a distrusted participant of the protocol. In a simulation-based proof, a distrusted party is challenged to distinguish between two constructions: the real protocol execution (with real inputs) and a *simulation* of it. Such simulation is not provided with the data of which privacy is desired. The key concept is that if a distinguisher is not able to tell the real protocol and the simulation apart, then the execution of the protocol cannot leak any information about the private inputs of the real parties[2].

[2]Information leaked by the output of the protocol is not considered as leaked.

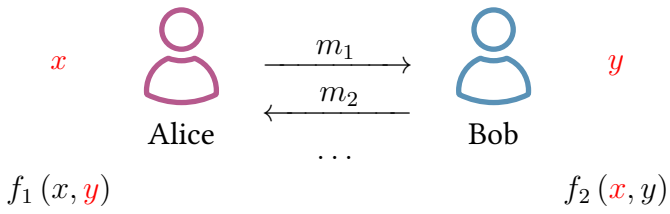
Secure computation

An important class of protocols in cryptography are secure computation protocols, or secure function evaluation protocols. By employing a secure computation two or more distrusting parties may compute a function of their private input that shall remain private. Secure computation embraces network communication protocols adopted in a large variety of real world contexts, such as secure communication in the Internet, e-commerce and banking, cloud computation, WiFi an mobile phone networks, proximity networks (NFC, RFID, Bluetooth, and others), blockchain technologies and e-voting. More precisely, no information about the input data shall be leaked beyond what is expected to be inferred by knowing their own private input and the final output produced at the end of the protocol[3].

The difference between secure computation and traditional cryptography is that privacy is to be preserved against distrustful participants rather than against entities external to the protocol. This difference on the adversarial model does not split apart secure computation and traditional cryptography: for example when two participants in a n-party protocol interact, if an attack revealing the private key hold by one participant can be found with traditional cryptography, then the same attack might be carried out by any distrustful participant of the protocol.

An example of secure computation, restricted to two parties is illustrated in Figure 2.2. In secure computation, the main focus is to protect participants' privacy

Fig. 2.2 Two-party secure computation scheme.



from each other. Proofs in secure computation are generally structured in a simulation-based approach, where honest parties are simulated without the parties' input. The privacy is captured by an indistinguishability theorem, where the corrupted parties cannot tell the real protocol and the simulation apart; therefore, the protocol does not leak any information about parties' inputs.

Formal definitions

In such a setting, privacy has to be protected against adversary who can corrupt participants and can control the communication channel. The simulator has to simulate the execution with corrupted participants as the alternative game to be distinguished from the real execution by the distinguisher. To do so, the simulator needs to emulate the *view* and the result of the protocol from the eyes of the corrupted party (without

[3]Or at some point of the protocol, e.g. in commitment schemes the input is eventually revealed, but it is crucial to protect its privacy up to the revealing phase

knowing the honest party's private input). We adapt our Definition 2.2.1 of view from Hazay and Lindell [99].

Definition 2.2.1 (View). *Let P_i be the i-th party executing the protocol Π run by $N \geq 2$ participants. Then the view of P_i in the protocol Π, denoted as $\mathsf{view}_{P_i}^{\Pi}$, is defined as*

$$\mathsf{view}_{P_i}^{\Pi}(x, n) \stackrel{\text{def}}{=} (x, r, m_1, m_2, \ldots, m_j)$$

where x_i is P_i's input, r is P_i's internal random tape, $m_1, m_2, \ldots, m_j$ are the messages received by P_i during the execution of Π, $x = (x_1, x_2, \ldots, x_N)$ are the parties' inputs, and n is the security parameter.

Notation 2.2.1 (Output). *We denote the output of a P_i in the protocol Π as $\mathsf{output}_{P_i}^{\Pi}$, and the joint output of all $N \geq 2$ parties as*

$$\mathsf{output}^{\Pi}(x, n) \stackrel{\text{def}}{=} \left(\mathsf{output}_{P_1}^{\Pi}(x, n), \mathsf{output}_{P_2}^{\Pi}(x, n), \ldots, \mathsf{output}_{P_N}^{\Pi}(x, n)\right).$$

where $x = (x_1, x_2, \ldots, x_N)$ are the parties' inputs and n is the security parameter.

From this point on, we restrict our definitions to two-party protocols, even though the same concepts can be extended to multi-party protocols, whose discussion is out of our scope. If we want, for example, to capture the security of a secure computation protocol with respect to *semi-honest adversaries*, we can require that the probability ensemble emulated by the simulator and the functionality result to be computationally indistinguishable from the probability ensemble of the view and the output of real executions of the protocol, as captured by Definition 2.2.2 by [99]. Semi-honest, or honest-but-curious, adversary is one who runs the protocol honestly but tries to infer as much information as possible from running the protocol.

Definition 2.2.2 (Security against semi-honest adversary). *We say that the protocol Π, run by two parties,* securely computes *a functionality $f = (f_1, f_2)$ in the presence of* semi-honest *adversaries if there exist two polynomial-time simulators $\mathcal{S}_i$, such that for all $i \in \{1, 2\}$, we have*

$$\{\mathcal{S}_i(1^n, x_i, f_i(x)), f(x)\}_{x,n} \stackrel{\text{c}}{\equiv} \left\{\mathsf{view}_{P_i}^{\Pi}(x, n), \mathsf{output}^{\Pi}(x, n)\right\}_{x,n}$$

where $x = (x_1, x_2)$ are the parties' inputs, the simulator $\mathcal{S}_i$ emulates the behaviour of the N parties excluding the adversarial P_i, $x = (x_1, x_2)$ are the parties' inputs, and n is the security parameter.

Simulating an indistinguishable view and output of the protocol does not capture capabilities of a malicious adversary. To capture such stronger notion of security, we require that adversary in an *ideal model* are able to simulate executions of the real protocol. The ideal model provides the intended functionality of the real protocol by mean of a trusted third party. We consider two-party protocols as our discussion involve only two-party protocols. In the ideal model, we have the honest party, the malicious party, and a trusted third party. The adversary sends inputs to the trusted party, who sends the result back to all parties. This last operation can be prevented

by the adversary instructing the third party to abort. At the end, the honest party outputs the value it received from the trusted party, while the adversary output the input of the honest party, the input she used (that may or may not equal the original one from the corrupted party), and the result obtained from the trusted party. So, intuitively, if the adversary cannot do more harm in the real protocol than it would in the ideal model, then the real protocol is secure.

Notation 2.2.2. *Let f be a functionality and Π a two party protocol for computing f. We denote the* real execution *of Π on inputs (x, y), in the presence of the adversary $\mathcal{A}$ that corrupts the party P_i and uses auxiliary input z, as*

$$\mathsf{Real}_{\Pi,\mathcal{A}(z),P_i}(x, y, n)$$

where n is the security parameter. Similarly, we denote the ideal model *of f on inputs (x, y), in the presence of the adversary $\mathcal{A}'$ that corrupts the party P_i and uses auxiliary input z, as*

$$\mathsf{Ideal}_{f,\mathcal{A}'(z),P_i}(x, y, n).$$

We report the definition of secure two-party computation by Hazay and Lindell [99] with the notation above.

Definition 2.2.3 (Security against malicious adversary)**.** *Let f be a functionality and Π a two party protocol for computing f. The protocol Π, run by parties P_0 and P_1, is said to* securely computes *the functionality f in the presence of* malicious *adversaries if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real execution there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{A}'$ in the ideal execution, such that for all $i \in \{0, 1\}$, we have*

$$\mathsf{Real}_{\Pi,\mathcal{A}(z),P_i}(x, y, n) \stackrel{\mathrm{c}}{\equiv} \mathsf{Ideal}_{f,\mathcal{A}'(z),P_i}(x, y, n).$$

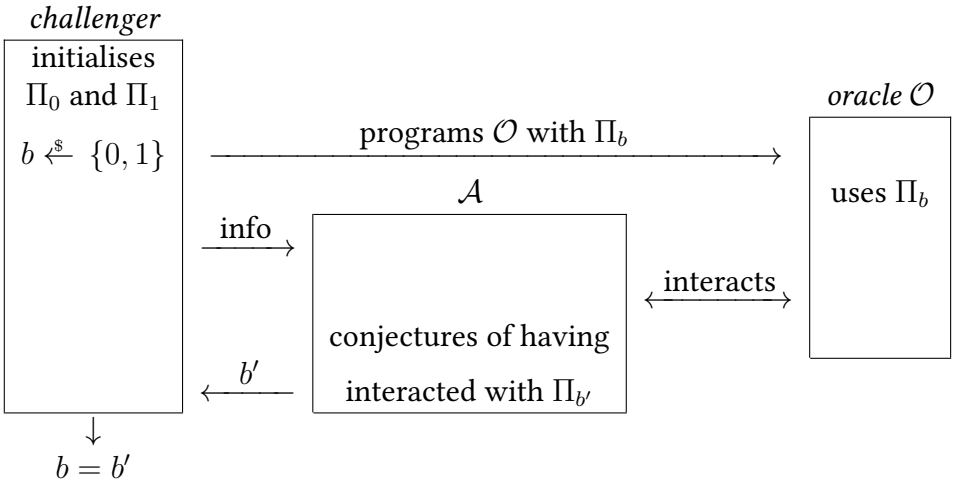
where $\mathcal{A}$ and $\mathcal{A}'$ corrupt the party P_i and use auxiliary input z, and n is the security parameter.

2.2.4 Oracles

An oracle is an entity that provides *black-box* access to functionalities. The caller (including the adversary) provides inputs, and the oracle will always provide the corresponding output without leaking any information about its internal procedures. In the context of indistinguishability of two protocols Π_0 and Π_1, an adversary is allowed to call an oracle which can be programmed with either Π_0 or Π_1. The adversary is finally challenged to determine what protocol was used to program the oracle.

The astute reader could argue that this behaviour can be actually captured by providing the adversary with the *full* description of the functions along with the transcripts of the protocol. However a more astute reader would also consider that the full description of the functions may have super-polynomial length (e.g. a random function [109]), and the (polynomial) adversary would not even have enough time

Fig. 2.3 Flow of the experiment $\mathsf{oExp}^{\mathcal{A},\mathcal{O}}_{\Pi_0,\Pi_1}$ for indistinguishability between a protocol described by the constructions Π_0 and Π_1, where the adversary $\mathcal{A}$ is allowed to interact with the oracle $\mathcal{O}$ at most a polynomial number of times.



to read its input. In the case when this problem is to overcome, the definitions of security are slightly modified to give the adversary access to an *oracle*.

The adversary is not provided with an output generated by a run of a protocol (or a simulation of it), but she interrogates the oracle upon chosen input at most a polynomial number of times. The oracle masks the internal behaviour, so it can be either the real execution of the protocol or not, without the adversary knowing. The indistinguishability game $\mathsf{oExp}_{\Pi_0,\Pi_1}$, illustrated in Figure 2.3 relating the two construction Π_0 and Π_1, is slightly different from the experiment illustrated in Figure 2.1, and it can be described as follows:

1. The challenger flips a coin to program the oracle to use either Π_0 or Π_1.
2. The challenger initialises the protocols in Π_0 and Π_1 and provides some related information to the distinguisher.
3. The adversary is allowed to interact with the oracle $\mathcal{O}$ a polynomial number of times[4], then sends her best guess to the challenger.
4. Finally, the challenger outputs whether the adversary guesses correctly or not.

Definition 2.2.4 (Computational indistinguishability with oracles)**.** *Given an indistinguishability experiment* $\mathsf{oExp}^{\mathcal{A},\mathcal{O}}_{\Pi_0,\Pi_1}$ *over two (terminating) constructions* Π_0 *and* Π_1 *run against the adversary* $\mathcal{A}$*, the probabilistic ensembles* $\mathcal{X}$*, due to interaction with an oracle* $\mathcal{O}$ *programmed with* Π_0*,* $\mathcal{O}_0$*, and* $\mathcal{Y}$*, due to interaction with* $\mathcal{O}$ *programmed with* Π_1*,* $\mathcal{O}_1$*, are* computationally indistinguishable*, denoted as* $\mathcal{X} \overset{c}{\equiv} \mathcal{Y}$*, if and only if for any probabilistic polynomial-time adversary* $\mathcal{A}$ *exists a negligible function* μ *such that*

$$\mathbf{Adv}^{\mathsf{oExp}}_{\Pi_0,\Pi_1}(1^n) = |\mathsf{Pr}[\mathcal{A}(\mathcal{O}_0) = 1] - \mathsf{Pr}[\mathcal{A}(\mathcal{O}_1) = 1]| \leq \mu(n)$$

[4]The adversary gets no extra information from the oracle if not the output of the interrogated functions.

where n is the security parameter, and $\mathcal{A}$ has no access to the internal state of the oracle, which is programmed uniformly at random with Π_0 or Π_1.

For simplicity, the constructions themselves are said to be indistinguishable, explicitly omitting reference to both their probabilistic ensembles and the oracles, with the notation

$$\Pi_0 \stackrel{c}{\equiv} \Pi_1.$$

Despite the notation is identical, this definition is different from the Definition 2.1.3 in the fact that the adversary is not directly provided with the ensemble generated by running Π_b; in fact, the adversary constructs the ensemble by interacting with the oracle $\mathcal{O}$.

As the reader may already have noticed, the Definitions 2.2.3 and 2.2.4, defining respectively a secure computation protocol and computational indistinguishability, are related. In particular, the oracle $\mathcal{O}$ in the latter can be programmed either with the real protocol, defining the Real execution in the former, or with the simulation, defining the Ideal execution in the former. Hence, the indistinguishability between the Real and the Ideal executions in the former definition relates to the indistinguishability of the constructions Π_0 and Π_1 in the latter definition.

2.2.5 Adaptive adversary

We will define the security of searchable encryption schemes, see Chapter 6, against *adaptive* adversary. In contrast to non-adaptive adversaries where the strategy (i.e. inputs and queries) are chosen beforehand, i.e. before the cryptographic experiment runs, an adaptive adversary can change its strategy during the execution of the protocol: in particular, the adversary can choose inputs and procedure calls depending on partial output of the protocol. We note that the adversary in the indistinguishability Definition 2.2.4 is allowed to interact with the oracle without a prescribed strategy and therefore make its next step of strategy dependent from the (partial) output obtained by interacting with the oracle. Thus, Definition 2.2.4 can be used to model security against adaptive adversaries[5]. Petcher and Morrisett [137] mechanised security against non-adaptive adversary of a searchable encryption scheme, and they note that, to tackle the adaptive version, they would need to employ oracles.

2.3 Formal models in Cryptography

To formally prove security properties of protocols, cryptographer adopt mathematical models. Cryptographic models can be intuitively seen as points of view to describe the same protocol and to capture different aspects of it. In this context, the security analysis of a cryptographic protocol is seen as the reasoning process over a protocol specification, given an adversarial model and initial assumptions. In brief, the mathematical description of the protocol, over which the reasoning is carried out,

[5]The same definition can model non-adaptive adversaries too, depending on if the adversary is additionally restricted to choose input and queries beforehand or not.

is the cryptographic model. It is important to stress that every such a model is an abstraction of the real settings where the protocols actually run.

Several models have been proposed in the literature to reason about cryptographic protocols, but two became more popular than others amongst cryptographers: the *symbolic model* [76] and the *computational model* [89], both dating back to nearly 40 years ago. In a nutshell, the symbolic model assumes the security of cryptographic primitives and uses them as black-boxes, so that the focus is shifted to the composition of the (parts of the) protocols; conversely, the computational model allows for reasoning about the mathematical details of cryptographic primitives and therefore their security properties. More details about the two models are presented in Sections 2.3.1 and 2.3.2. A through discussion of their differences is illustrated in Appendix A.3, where we show a formalisation of asymmetric encryption through the lens of either of the models.

The need for automated reasoning

The symbolic and the computational model have been used in on-paper proofs of cryptography for long time. In the past decades, research and industry have witnessed an escalation in the amount of cryptographic constructions, and consequently the mathematical proofs of their security claims increased in both amount and complexity. Their review, often by peers, became a very time consuming process, and attacks were found to protocols believed to be secure for decades [120]. More importantly, such incidents were evidence that cryptography was (and is) suffering from a lack of rigour.

The response of the research community to the lack of mathematical rigour in cryptography have been that of delegating some of the reasoning process to machines, by means of automatic tools. Automated tools mitigate such issues and aid the reasoning about cryptographic protocols, with the concomitant benefit of speeding up the reviewing process. In particular, they started implementing automated tools to aid the formal verification of security protocols that handle either the computational or the symbolic model. Those tools significantly reduce the amount of trust to put in on-paper proofs, as they cover all gaps that are commonly assumed as *obvious* to simplify the discussion in the proofs. Those mechanised proofs are easily reproducible using a machine.

The Sections 2.3.1 and 2.3.2 will introduce two reasoning models that are used to represent reality: they balance efficiency, expressiveness, completeness and generality to reach different degrees of automation.

2.3.1 Symbolic model

The symbolic model was introduced by Yao in 1983 [76], but literature witnesses its informal usage even before [134]. It describes cryptographic protocols abstractly. On one hand, its high level of abstraction allows to make tools for automatic reasoning. On the other hand, its expressibility is limited and potentially hard-to-apply to realistic scenarios. In particular, (i) messages are literals, that simplifies the reasoning on their properties, like equality; (ii) cryptographic primitives are assumed to be

perfect and used as black boxes, that allows to focus on other properties, like the composition of protocols; and (iii) the adversarial strategies are predefined and limited to the inference rules provided, that on one side it limits the proofs to case-by-case reasoning [18], but on the other side it makes the automation easier. In this model, adversaries can handle the messages through insecure channels, that is they can eavesdrop, intercept, deliver, block or tamper with messages.

The symbolic model can capture errors in the logic of the design, but cannot fully describe situations where the cryptographic primitives cannot be treated as black boxes or some cases when the security properties are defined computationally.

Most of the automatic tools aiming to aid proofs for cryptographic protocols [111, 129, 49, 143, 66, 14, 30, 158, 79, 139, 124, 12, 144] work in the symbolic model, some providing computational soundness for special cases [17].

2.3.2 Computational model

Automatic tools supporting the computational model [45, 26, 11] have been developed later than those in the symbolic model, due to their inherent more detailed description of the real world, hence their complexity.

Differently from the symbolic model, the computational model is closer to complexity theory, e.g. algorithms are Turing machines with random tapes, when modelling cryptographic protocols and can capture many low level details which are needed in proofs. This offers the possibility to reason with the principles on which the machines are built: messages are bit strings, and algorithms and adversaries are probabilistic algorithms. Mathematical theories are first-class citizens in the computational model, which therefore allows for reasoning with probabilities, complexity, and cryptographic assumptions. This allows for rigorous proofs that cannot be obtained in the symbolic model.

Security is defined against efficient adversaries can break the security properties with only a negligible probability, if certain assumptions hold.

Due to the many mathematical details, theories, and lemmas, the computational model shows harder proofs and it is generally more difficult to automate. This is usually reflected in the huge amount of effort (thousands of lines of code[6]) that hardly scale or lead to reusable code [56].

2.4 Reasoning in the Applied Pi Calculus in ProVerif

ProVerif [49] is a tool for reasoning in the symbolic model. It has proved successful in formally verifying dozens of protocols, and has been widely accepted by the community. ProVerif's input language is a dialect of the Applied Pi Calculus [50, 5], and we limit our informal illustration of the language to the subset that will be useful for our model of the SPEKE protocol that we will discuss in Chapter 3.

The language is strongly typed, and arbitrary types can be declared. Functions are modelled by *constructors* and *destructors*. Constructors are abstractions of functions

[6]We remark that the most of the code is used by mathematical proofs in theorem provers.

whose signature is their name, their arity, and in typed languages the type of the arguments and the returning type. Constructors with arity 0 model constants, while constructors with higher arity model generic functions. Destructors can be roughly considered as *inverse* of constructors and determine reductions applied to constructors; for example, the decryption function is a destructor for the encryption function as its application to an encryption of a message *reduces* to the mere plain message (as long as the decryption key is correct). To model other properties, as commutativity, associativity, or other properties, additional equations can be provided[7].

The basic syntax of the language is provided in Figure 2.4, adapted from the manual of ProVerif. Its basic elements are called *terms* to which expressions can

Fig. 2.4 ProVerif's language syntax for processes (Applied Pi Calculus). E are expressions, M patterns, and P processes, illustrated in the Backus-Naur form. Every *else* branch and *suchthat* condition are optional. The nil process $\mathbf{0}$ may be implicit.

$E ::=$	$a, b, c, \ldots$	names (atomic data), variables, tables
$\mid$	$f\,(E, E, \ldots, E)$	function application
$\mid$	$(E, E, \ldots, E)$	tuples (built-in application)
$\mid$	$\neg E$	negation
$\mid$	$E \Diamond E$	binary function application ($\Diamond$ can be $=$, $\neq$, $\wedge$, or $\vee$)
$M ::=$	$x : T$	bind to variable x of type T
$\mid$	$= E$	equality test
$\mid$	$f\,(M, M, \ldots, M)$	data constructor with patterns
$P ::=$	$\mathbf{0}$	nil
$\mid$	$\text{in}\,(E, M)\,; P$	input from channel E of pattern M
$\mid$	$\text{out}\,(E, E)\,; P$	output an expression to a channel
$\mid$	$P \mid P$	parallel composition
$\mid$	$!P$	infinite replication
$\mid$	$\nu a.P$	restriction
$\mid$	$\text{if } E \text{ then } P \text{ else } P$	conditional
$\mid$	$\text{let } M = E \text{ in } P \text{ else } P$	evaluation
$\mid$	$\text{insert } t\,(E)\,; P$	insert E to table t
$\mid$	$\text{get } t\,(M) \text{ suchthat } E \text{ in } P \text{ else } P$	read from table t
$\mid$	$\text{event } e\,(E)\,; P$	record event e with arguments E

be applied: terms are expressions too. The available expressions allow for equality and inequality (just the not-equal) of terms, conjunction and disjunction, negation, pairing (in tuples), and function application. A name is atomic data, while a variable can be substituted by any term. Tables are particular extensions of the language which in the Applied Pi Calculus are absent; however, it is easy to show how *restricted* channels[8], appropriately used, can act exactly as such tables. Both constructors and

[7] Adding equations generally comes at the price of slowing significantly the reasoning process as more cases must be analysed.

[8] restricting a channel to processes that model entities makes such a channel private to those entities; otherwise, by default a channel is accessible by the attacker.

destructors are function symbols, but in the tool they determine different sets of rules and semantics.

The main abstraction of the Pi Calculus is the process. A process describes the algorithm that an entity follows according to the specifications of a protocol scheme. They can (i) include variables, constants, functions, and (private) nonces (i.e. $\nu x.P$ restricts the value x to the process P) (ii) write to and read from any channel c, denoted by $\mathrm{out}\,(c, \_)$ and $\mathrm{in}\,(c, \_)$ respectively, (iii) insert and extract elements to and from any table t, $\mathrm{insert}\ t\ (\_)$ and $get\ t(\_)$, and (iv) record events. Processes can be put in sequential or parallel execution with other processes including themselves, with implicit barriers at $\mathrm{in}\,(\_, \_)$ operations, for unbounded number of times of replication. Such instruments allow for symbolic modelling of protocols. In fact, the parties can send messages to the others, can instantiate (private) nonces and do some operations whose result can be used later on in the protocol, can check for validity of received messages, and can receive external values before engaging the protocol.

To model security properties, the language offers some facility. The secrecy of names is verified in terms of unreachability and indistinguishability. The unreachability of the secret by the attacker determines whether the knowledge of the attacker can be augmented with such secret by using the inference rules determined with respect to the model. From the point of view of indistinguishability, the tool determines whether the attacker can distinguish between executions that use different secrets.

More sophisticated security properties, like entity authentication, bilateral unknown shared-key resilience, and others, can be formalised through events and correspondences [46]. Events must be explicitly included as extra lines into the processes, can take arguments, and will be recorded in the traces of execution. Correspondences are implications related to execution of events. By default the content of events is not accessible to the attacker, until the attacker is already aware of them or it will be by other rules. Moreover, the attacker is not directly capable of recording events, but it may induce processes to do so.

For example, we want to capture authentication. Then, an event $e_1\,(A, B)$ can be inserted through the lines of the process and can be interpreted as "Alice believes of having *started* an authentication with Bob"[9]. Similarly, an event $e_2\,(A, B)$ can be interpreted as "Alice believes of having *completed* an authentication with Bob". Thus, authentication can be defined as a relationship between the events e_1 and e_2, $e_2\,(A, B) \Rightarrow e_1\,(A, B)$. The interpretation of $e_2\,(A, B) \Rightarrow e_1\,(A, B)$ as authentication property is that the event e_2 must always be preceded by e_1 in any possible trace records, so the overall interpretation would be "Whenever Alice believes of having *completed* an authentication with Bob, then the authentication procedure was actually *started* by Alice authenticating Bob" or, in other words, "Alice is sure that she's speaking to Bob".

The reasoning engine of ProVerif will execute a *main* process and record traces of execution. At the same time, the attacker's knowledge and the tables, if any, are accordingly updated. The reasoning core of ProVerif will try and compute all

[9] As a curiosity, capturing *belief* is typical of the BAN logic [7], but events generalise them to interpret other kind of statements related to the purpose of the protocol. Events have been introduced on top of a criticism to a lack of formality in the BAN logic [161].

(infinite) traces of execution and, where possible, it applies theorems for pruning allowing for reasoning about unbounded number of executions of the processes[10]. Security properties or attacks are eventually checked by inspecting traces, tables, the attacker's knowledge, and, for equivalences, relations between traces and processes. We refer to the paper by Blanchet [49] for additional details.

2.5 Reasoning in the probabilistic Relational Hoare logic in EasyCrypt

EasyCrypt is a tool for reasoning in computational model and imperative code. To do that it implements different logics [21]: apart from higher-order logic (HOL), it embeds Hoare logic (HL) to allow for reasoning about pre and post conditions of procedures' execution, a probabilistic Hoare logic (pHL) which allows for reasoning on probabilities of pre and post conditions, and a probabilistic relational Hoare logic (pRHL) to relate two procedures. We call *judgement* any proposition in HL, pHL and pRHL. To better illustrate facts later in the dissertation, we need to borrow some formula from the pHL [43] and pRHL [26, 24].

Let us consider a probabilistic[11] procedure c, declared in the module M, and a pre and a post condition Ψ and Φ. We denote by $A_{[m_1,m_2,\ldots]}$ the validity of the relation A whose propositions can relate to the memories m_1, m_2, and so on; we also denote by $c\,[m]$ that the procedure c runs in the memory m. We use a pHL judgement to relate the probability of the post condition Φ to be true after running c with precondition Ψ to a real number e; we use the notation

$$\models M.c : \Psi \Rightarrow \Phi \lozenge e$$

where e is a real-typed expression and $\lozenge$ is a logic operation among $<$, $\leq$, $=$, $\geq$, and $>$, and is valid for all memories m. One example is termination. Termination is formally expressed as *losslessness*, that is the probability that the post condition $\mathtt{T}$ (true) holds after running c is 1.

$$\models M.c : \mathtt{T} \Rightarrow \mathtt{T} = 1$$

To relate two procedures, we use pRHL judgements. Let us consider the probabilistic procedures c_1 and c_2 declared in the modules M_1 and M_2 respectively, and a pre and a post condition Ψ and Φ. The conditions are binary relations that can refer to memories where the procedures run. Rather than having a single procedure $M.c$, we have the relation between the procedures as $M_1.c_1 \sim M_2.c_2$. We say that the judgement denoted by

$$\models M_1.c_1 \sim M_2.c_2 : \Psi \Rightarrow \Phi$$

[10] The reasoning may not terminate, as discussed in Section 2.6.

[11] Technically, it is expressed in the pWHILE language.

is valid if for all memories m_1 and m_2 as environments for c_1 and c_2 respectively, the validity of $\Psi_{[m_1,m_2]}$ before the execution implies that a proper transformation of $\Phi'_{[m'_1,m'_2]}$ where the memories m'_1 and m'_2 have been modified by the execution of c_1 and c_2 respectively. This judgement is helpful to reason about probability of cryptographic games with the following rule.

$$\frac{\models M_1.c_1 \sim M_2.c_2 : \Psi \Rightarrow \Phi \qquad \Phi \Rightarrow A_{[m_1]} \Rightarrow B_{[m_2]}}{\mathsf{Pr}\left[M_1.c_1\left[m_1\right] : A\right] \leq \mathsf{Pr}\left[M_2.c_2\left[m_2\right] : B\right]}$$

Or similarly, for equality, we have

$$\frac{\models M_1.c_1 \sim M_2.c_2 : \Psi \Rightarrow \Phi \qquad \Phi \Rightarrow A_{[m_1]} \Leftrightarrow B_{[m_2]}}{\mathsf{Pr}\left[M_1.c_1\left[m_1\right] : A\right] = \mathsf{Pr}\left[M_2.c_2\left[m_2\right] : B\right]}$$

As the reader may have already noticed, the pRHL naturally allows for capturing the definitions of security and reason about the proof structure illustrated in Section 2.2.

In the pRHL, we can also describe computational equivalence up to a *bad* or failure events whose probability can be reduced to negligible or to cryptographic assumption, as introduced in Section 2.2.2. The rule to apply to the proof in this case is more complex than the equivalence described above. In particular, two algorithms are equivalent if an invariant can be stated when calling the distinguisher $\mathcal{A}$. The invariant must hold in the case that every oracle function is accessed and in the case those functions are never used. Plus, we must prove the termination of the distinguisher. More formally, the $\mathcal{A}$ is asked to distinguish between two constructions Π_1 and Π_2, running in memories m_1 and m_2. $\mathcal{A}$ is not directly given the construction, but can access to its functionality through an oracle $\mathcal{O}$, whose procedures are programmed with either of the constructions. Moreover, $\mathcal{A}$ has no access to the internal state of the oracle. So, assuming that the precondition Ψ holds, the indistinguishability of the two constructions up to the bad event b (that can happen in the rightmost construction Π_2) can be written as:

$$\frac{\begin{array}{c}\forall i.\ \models o_i \sim o_i : \neg b \wedge I \wedge =\{a_i\} \Rightarrow (\neg b \Rightarrow I) \wedge =\{r_i\} \\ \forall i.\ \forall m.\ b\left[m\right] \Rightarrow o_i : \mathtt{T} \Rightarrow \mathtt{T} \\ \forall i.\ \models o_i : b \Rightarrow b \\ \models \mathcal{A}\left(\mathcal{O}_{\Pi_1}\right) \sim \mathcal{A}\left(\mathcal{O}_{\Pi_2}\right) : \Psi \Rightarrow (\neg b \Rightarrow I \wedge =\{\mathcal{A}\}) \\ \wedge \forall r_1\ r_2\ \mathcal{A}_1\ \mathcal{A}_2,\ (\neg b \Rightarrow =\{r\} \wedge \mathcal{A}_1 \equiv \mathcal{A}_2) \Rightarrow \neg b \Rightarrow r_1 \Rightarrow r_2\end{array}}{\left|\mathsf{Pr}\left[\Pi_1\langle m_1\rangle = 1\right] - \mathsf{Pr}\left[\Pi_2\langle m_2\rangle = 1\right]\right| \leq \mathsf{Pr}\left[b\right]} \tag{2.1}$$

where $o_i\left(a_i\right)$ are n callable procedures whose arguments are a_i, I is an invariant relation that can involve the memories m_1 and m_2, $=\{\bullet\}$ denotes equality of the same variable or internal state in both memories, and r_i is the return value of the distinguisher. The goals above capture the following concepts respectively: (i) for all called procedures with the same arguments, if the bad event b do not happen, then the procedures show the same result (statistically speaking), and the invariant I is respected before and after the execution; (ii) for all memories, the occurrence of the

bad event implies that all callable procedures terminate; (iii) for all called procedures, they never reset the bad event to F (false); (iv) if the bad event does not occur, then the invariant I is respected and the result of the distinguisher provided with $\mathcal{O}_{\Pi_2}$ is the same as the result of the distinguisher provided with $\mathcal{O}_{\Pi_1}$; moreover for all results and adversaries, the absence of the bad event would put in relation the results in such a way that the first result entails the second. More informally, the last point states the indistinguishability when the bad event will not occur in the postcondition. The above is one of the core rules to prove computational indistinguishability through the advantage using oracles.

EasyCrypt

We present a brief overview of the EasyCrypt language, based on the pRHL. More information about Easycrypt and the syntax of its language can be found in [21, 3]. Those who are familiar with EasyCrypt can skip the rest of this section.

EasyCrypt handles the computational model, in which adversaries are probabilistic algorithms. To capture this, we have modules that are containers of global variables and procedures. Procedures capture the idea of algorithm, and one can reason about procedures running in a memory (as an execution environment). In the computational model, one has to reason about the probability of adversaries returning some specific results. EasyCrypt captures this idea as the probability of running a procedure $M.c$ in a memory m with post-condition Q evaluating true, where M is the module containing the procedure c, written as

$$\texttt{Pr[}M.c\texttt{(...) @ \&}m\ \texttt{:}Q\texttt{]}.$$

To express and to prove properties, EasyCrypt supports judgements (assertions) in (i) basic higher-order logic for implemented theories, (ii) Hoare logic (HL), (iii) probabilistic Hoare logic (pHL), and (iv) probabilistic relational Hoare logic (pRHL). For the last three of them, there are concepts of pre-condition P and post-condition Q, as well as procedures $M.c$, $M.c_1$, $N.c_2$, ..., inside modules M, N, running in some memory m. The post-condition Q of the probability expression and the three judgements can relate to a special term `res` identifying the return value of the procedures involved.

HL `hoare[`$M.c$ `:`$P \Rightarrow Q$`]` - When P is true relating to some memory m and $M.c$ terminates in m, then after running $M.c$, Q always evaluates to true in the (modified) memory.

pHL `phoare [`$M.c$ `:`$P \Rightarrow Q$`] <` r - When P is true relating to some memory m and $M.c$ terminates in m, then after running $M.c$, Q evaluates to true in the (modified) memory with probability less than $r \in [0, 1] \subset \mathbb{R}$. Other supported relations are the common relations $=$, $>$, $\geq$, and $\leq$.

pRHL `equiv[`$M.c_1 \sim N.c_2$ `:`$P \Rightarrow Q$`]` - When P is true relating to some memory m and $M.c_1$ and $N.c_2$ terminate in m, then after running them in two separate copies of m, Q always evaluates to true in the corresponding memories.

In cryptography, security is usually defined by requiring certain properties to hold for all adversaries. To capture the *for all* quantifier, in EasyCrypt, adversaries are defined with abstract procedures, which means the adversaries can do anything without any prescribed strategies. Working with abstract procedures may require to assume their termination. In Easycrypt, the idea of termination is modelled by the keyword `islossless`. We can declare a procedure to be *lossless* using the following syntax:

$$\texttt{islossless}\ M.c.$$

The statement is defined as a pHL judgement `phoare[`$M.c$ `: T ⇒ T] = 1%r`, which means the procedure M.c always returns and terminates with a probability 1 (`1%r` means real number 1).

Other common construction for reasoning with Turing machines is judgements in the Hoare logic (HL), where roughly an algorithm c is put in the middle of a pre-condition P and a post-condition Q ($\{P\}\, c\, \{Q\}$). The meaning of the Hoare triplet is that if the precondition P is true in some environment (read memory) before running c, and if c will terminate its execution in the environment, then after running c the condition Q is desired to hold true. In EasyCrypt, being the procedure c in the module M, we would have

$$\texttt{hoare[}M.c\ \texttt{:}P \Rightarrow Q\texttt{]}$$

Another type of judgements is of type probabilistic Hoare logic (pHL), that is a Hoare triplet related to some expression r evaluating in the $[0, 1] \in \mathbb{R}$ domain. Loosely speaking, the meaning of $\{P\}\, c\, \{Q\} < r$ is: If the precondition P is true in some environment before running c, and if c terminates its execution in such environment, then after running c the condition Q will hold true with probability strictly less than r. Other supported relations are the common relations $=$, $>$, $\geq$, and $\leq$. In EasyCrypt, being the procedure c in the module M, we would have

$$\texttt{phoare [}M.c\ \texttt{:}P \Rightarrow Q\texttt{]}\ \texttt{<}\ r$$

The last type of judgement supported in EasyCrypt is of type probabilistic relational Hoare logic (pRHL). These judgements compare two algorithms c_1 and c_2 for the same precondition and post-condition ($\{P\}\, c_1 \sim c_2\, \{Q\}$) [43]. The meaning of a pRHL judgement is that if the precondition P is true in some environment before running both c_1 and c_2 , and if they terminate their execution their own copy of the environment, then after running them, the condition Q will hold in both respective separated environments with the same probability. In EasyCrypt, being the procedures c_1 and c_2 in the modules M and N, we would have

$$\texttt{equiv[}M.c_1 \sim N.c_2\ \texttt{:}P \Rightarrow Q\texttt{]}$$

Working with abstract algorithms may require to assume their termination. If we want to manipulate them, we often require them to be *lossless*, capturing the idea that the probability of returning whatever value is 1. We can formally define the termination property of an algorithm by stating that it is lossless with a pHL judgement where both the precondition and post-condition are simply true values:

$$\texttt{islossless}\ M.c \stackrel{\text{def}}{=} \texttt{phoare}[M.c\ :\ \texttt{T} \Rightarrow \texttt{T}]\ =\ \texttt{1\%r}$$

where `1%r` means 1 of type real.

All the above constructions are useful for our formalised proof. For example the Pedersen commitment scheme [136] we discuss in Chapter 4, we capture the security properties of correctness, perfect hiding and computational binding. In particular, we express correctness as a HL judgement, we compare the hiding experiment with an artificial experiment and then a pHL to finalise the proof, and we use a pHL judgement to compare the binding experiment to the discrete logarithm experiment.

2.6 Spectrum of the tools available

In this section we will very briefly explore the tools that we considered to base our work on. When our journey begun, we did not know any of the tools that we are going to mention. We discuss what characteristics of the tools persuaded us to eventually use ProVerif (for analysis in the symbolic model) and EasyCrypt (for analysis in the computational model).

Symbolic model

Automatising the reasoning in the symbolic model is based on the inspection of the state space determined by the formal description of the protocol (model checking). The main challenge is due to the infinite state space to explore. This fact is mainly due to the unbounded number of concurrent executions of the protocol (sessions), the number of parties to consider (honest parties and corrupted parties), and the unbounded message size. Comon-Lundh and Cortier [65] showed that the number of parties can be bounded depending on the security property to study (one for secrecy and two for authentication). For unbounded sessions, the security of protocols is generally undecidable [77], while finding attacks to protocols when the number of sessions is bounded is an NP-complete problem, regardless of the message size [140].

Therefore some tools limit the number of sessions exploring only a part of the state space: FDR (Failures Divergences Refinement Checker) [120] and SATMC (SAT-based Model-Checker) [15] use standard model checking techniques, Murϕ [129] expands FDR methodology by adding more constructs, e.g. symmetry reduction or reversible rules, Maude [73] adopts rewriting logic where multiple concurrency models can be expressed, CL-AtSe (Contraint-Logic based Attach Searcher) [64] [127] handles infinite states models, and OFMC (On-The-Fly Model-Checker) [29] combines the use of lazy data-types, as a simple way of building an efficient on-the-fly model checker for protocols with infinite state spaces, and the integration of symbolic techniques, for modeling a Dolev-Yao intruder. OFMC can give results even for unbounded number of sessions in some special cases. The above tools are generally suitable to model and find attacks; however, when attacks are not found, it does not mean that the protocols are secure, as attacks could lie in the unexplored part of the state space.

Other tools restrict their reasoning to subclasses of protocol or allowing user interaction in order to provide results with infinite sessions: Interrogator [128], NPA

(NRL Protocol Analyzer) [123] and its successor Maude-NPA [80], Athena [150], Cryptyc [90], ProVerif [49], TA4SP (Tree-Automata based Automatic Approximations for the Analysis of Security Protocols) [51], Scyther [70], Tamarin [125]. The major disadvantage of these tools is that they are incomplete, they may not terminate or require user interaction to guide the tool to the right direction. In some cases, they limit the number of sessions to guarantee termination, like Scyther [70].

All the above mentioned tools vary from the points of view of efficiency, expressibility and capability. As a natural consequence, efforts have been put to collect (some) of those tools by creating an intermediate language that could be lifted to the languages of the tools: examples are CAPSL (Common Authentication Protocol Specification Language) [74], AVISPA (Automated Validation of Internet Security Protocols) [160] and AVANTSSAR [12]. CAPSL's intermediate language exports to Maude, NPA, Athena and the verifier [127], AVISPA's intermediate language HLPSL [63] exports to SATMC, CL-AtSe, OFMC and TA4SP, AVANTSSAR's intermediate languages are HLPSL and others, and it exports to SATMC, CL-AtSe, OFMC.

In our work, we modelled the protocol SPEKE in the symbolic model. The protocol specification is based on the Diffie-Hellman key exchange, relying on the group theory, and its security is based on the computational hardness of the discrete logarithm. Those characteristics would require additional equations to model group theory and at the same time allow for modelling multiple attacks, such as impersonation, reply and malleability attacks, and security properties, such as secrecy and authentication. Modelling group theory in the symbolic model increases significantly the state space to explore, and the only tools that (partially) supported it were ProVerif and Tamarin[12]. We dedicated the same learning-time to those tools. What persuaded us to use ProVerif was its support a stronger notion of secrecy (through equivalence properties) as well as authentication; so the only novel application would have been malleability, which we modelled by mean of properties over events in the traces of execution. What discouraged us to use Tamarin was mainly the difficulty of the language, and the (subjectively) unnatural way of describing a protocol through atomic interrelated rules. Tamarin mixes many syntaxes together and *runs* regardless of typos or undeclared variables making it difficult to review our own work (as beginners); on the same time frame dedicated to other tools, we could not build enough confidence on the language to understand the synergy between its own constructs. To the best of our understanding, the main advantage of using Tamarin over ProVerif would have been its ability to deal with properties that depend on the precise state of agent sessions and mutable global state [125]. However, we did not have such properties to study in the SPEKE protocol.

Computational model

The automated tools to reason in the computational model [37] is generally based on sequences of cryptographic games. The proof is done as a *small-step* reduction until a final game whose probability is known is reached. The transformation from

[12]We also considered CryptoVerif to reach guarantees in the computational model, but we could not reach the same level of automation.

a game to another needs to be (at most) negligible. The tools we considered are: CryptoVerif [45], EasyCrypt [26], the Foundational Cryptographic Framework [138], CryptHOL [31].

CryptoVerif is a highly automated tool [45] where the protocols may be written in the pi calculus [5]. Even if its automation is very appealing, it applies to some proofs only; complex proofs deviating from the supported ones requires either interactive mode, very difficult to use, or changing the internal core of CryptoVerif. This fact discouraged us from using it. However it has been successfully used to prove security for the popular TLS [48].

EasyCrypt is a tool for reasoning about relational properties of probabilistic computations with adversarial code. Even though it does not provide automatic proofs, it allows to write both algorithmic code and functional to model protocols. In particular, we find the former very close to usual programming languages. So anyone familiar with an imperative programming language would understand the modelling (but not the proofs) in EasyCrypt.

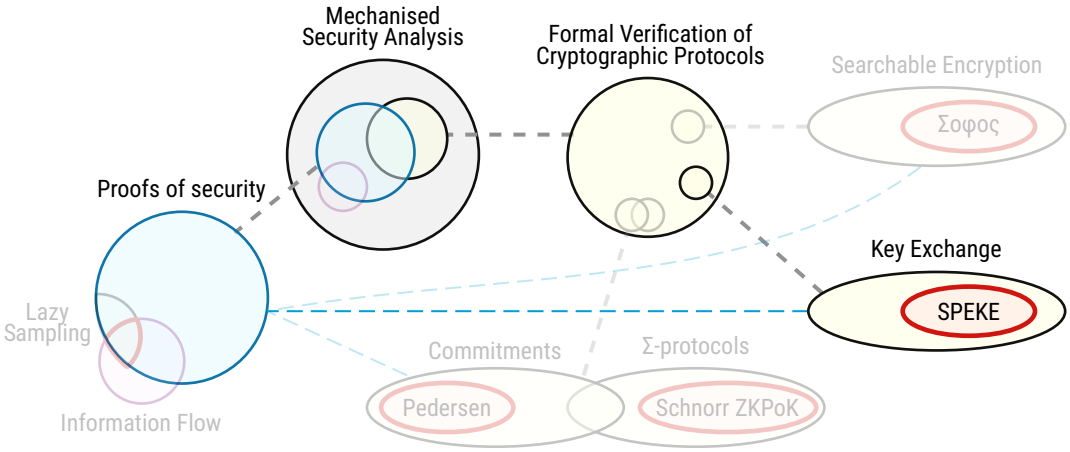
FCF works entirely under the theorem prover Coq, which is believed to be correct. However, it support only constructions in functional style and, in its current status, it could not handle security properties with oracles [138]. We needed to model searchable encryption security with oracles.

CryptHOL is based on Isabelle/HOL [135] and supports reasoning in the computational model [119]. It supports functional language description only, which we find much less natural than the code-based (imperative) approach of EasyCrypt that also resembles object orientied. Towards proofs of security, functional or imperative does not matter, as long they describe the same semantics; however, imperative languages are more likely to be adopted in real implementations, and therefore a non-expert in EasyCrypt will find the model very easy to understand and can appreciate how it reflects to actual code. Also, at the time we had chosen what tool to work with, CryptHOL was not mature enough to be considered, as EasyCrypt was already supporting game-based proofs and indistinguishability proofs (required for simulation-based proofs). Recent works [55] show how the tool has been extended to be a competitive choice to work with; in particular, the mechanisation of their extended trapdoor permutation has been done contemporarily with ours in EasyCrypt, its theoretical description is provided in Section 6.2.2.

Computational soundness can be obtained also on the basis of a symbolic analysis [6], if the cryptographic primitives satisfy strong security properties. This imposes restrictions to the protocol, de facto limiting the case studies. As our work investigates relaxed security properties, e.g. schemes that leak information, we opted to directly work in the computational model.

Chapter 3

Key exchange security in the symbolic model



In this chapter, we present a formal analysis of the key exchange protocol SPEKE. We carry out the analysis in the symbolic model with the tool ProVerif. We study many security properties that relate to confidentiality and authentication and model attacks to the scheme. Among the properties, we modelled malleability with events and discuss benefits and limits of this approach. We compare the security of several variants of the SPEKE protocol and propose a patch that has been included in the standard ISO/IEC 11770 [104]. Importantly, the SPEKE protocol has been verified in the past in the AVISPA tool [160], but their model was not capturing all the aspects as we do and could *not* find any attacks to the protocol.

3.1 Simple Password Exponential Key Exchange

Simple Password Exponential Key Exchange (SPEKE) is a well-known Password Authenticated Key Exchange (PAKE) protocol aiming to establish a high-entropy session key for secure communication between two parties based on a low-entropy secret password known to both without relying on any external trusted parties. The idea of bootstrapping a high-entropy secret key based on a low-entropy secret

password is counter-intuitive, and for a long time had been thought impossible until the seminal work by Bellovin and Merrit who proposed the first PAKE solution called Encrypted Key Exchange (EKE) [41]. Since then, research on PAKE has become a thriving field: many PAKE protocols have been proposed, and some have been included into international standards [100, 104].

However, the original EKE protocol was found to suffer from several limitations: the most significant one was the leakage about the password [106]. Motivated by addressing the limitations, Jablon proposed another PAKE solution called the simple password exponential key exchange (SPEKE) in 1996 [105]. SPEKE proves to be a more practical protocol than EKE since it does not have the same password leakage problem as in EKE. Although researchers raised concerns on some other aspects of SPEKE [163, 157] such as the possibility for an online attacker to test multiple passwords in one go, no major flaws have been reported. Over the years, SPEKE has been used in several commercial applications: for example, the secure messaging on Blackberry phones [2] and Entrust's TruePass end-to-end web products [1]. SPEKE has also been included into the international standards such as IEEE P1363.2 [100] and ISO/IEC 11770-4 [104].

Given the wide usage of SPEKE in practical applications and its inclusion in standards, we believe a thorough formal analysis of SPEKE is both necessary and important. In this chapter, we revisit SPEKE and its variants specified in the original paper [105], the IEEE 1363.2 [100] and ISO/IEC 11770-4 [103] standards. We first observe that the original SPEKE protocol is subtly different from those defined in the standards. The difference has significant security implications, which are not explained in the standards.

3.1.1 Password Authenticated Key Exchange

Since the invention of the first PAKE solution in [41], many PAKE protocols have been proposed, among which only a few have been actually used in practice. Notable examples of PAKE that have been deployed in practical applications include EKE [41], SPEKE [105] and J-PAKE [96]. These three protocols happen to represent three different ways of constructing a PAKE. EKE works by using the shared password as a symmetric key to encrypt Diffie-Hellman key exchange items. Variants of EKE, e.g. SPAKE2 [9], often differ only in how the symmetric cipher is instantiated. SPEKE works by using the shared password to derive a secret group generator for performing Diffie-Hellman key exchange. There are variants of SPEKE, such as Dragonfly [98] and PACE [42], which use different methods to derive the secret generator from the password. J-PAKE works by using the password to randomize the secret exponents in order to achieve a cancellation effect. A distinctive feature of J-PAKE as compared to the other two is its use of Zero Knowledge Proof (ZKP) [96] to enforce participants to follow the protocol specification. By comparison, the use of ZKP is considered incompatible with the design of EKE and SPEKE.

A PAKE protocol serves to provide two functions: authentication and key exchange. The former is based on the knowledge of a password. If the two passwords match at both ends, a session key will be created for the subsequent secure commu-

nication. In the following, we review some common properties of a secure password authenticated key exchange protocols based on [41, 105, 96]; we also refer the reader to classic definitions of authentication from Lowe [121]. Formal treatments of PAKE, based on authenticated key exchange models proposed by Bellare and Rogaway in 1993 [36], can be found in [35, 88, 110, 8].

Correctness. In the setting of key-exchange protocols, the protocol is correct if it gives both authentication and key distribution in presence of honest parties [161]. This is a basic and necessary step in a formal model to prove that without influence of attackers, honest parties should always complete the protocol as expected.

Secrecy of the pre-shared password. This property requires that the execution of the protocol must not reveal any data that would allow an attacker to learn the password through off-line exhaustive search. If the attacker is directly engaging in the key exchange, he should be limited to guess only one password per protocol execution.

Implicit key authentication. Assume the key exchange protocol is run between Alice and Bob. The protocol is said to provide implicit key authentication if Alice is assured that no one other than Bob can compute the session key [153].

Explicit key authentication. Explicit authentication can only be achieved with a confirmation phase [153]. This property requires that the entities have actually computed the *same* key. It completes and strengthens the implicit key authentication; in fact, if the two participants are the sole entities who can learn the session key *and* they have actually computed the key, the successive communication shall be secure.

Weak and strong entity authentication. *Weak* or *strong* entity authentication respectively correspond to the *weak agreement* and *injective agreement* properties of Lowe [121]. A protocol achieves weak authentication if a participant believes she is speaking with another participant, and the other participant indeed started an authentication process with her. Even though this may seem a sufficient property for mutual authentication, it is not. In fact, nothing can be said about the problem where the party is tricked to communicate with some replayed session of the other party. With strong authentication, the additional property of agreeing with both the session and the session key is required. Strong entity authentication ensures that replay attacks and man-in-the-middle attacks are prevented.

Perfect forward secrecy. Perfect forward secrecy (PFS) ensures that the confidentiality of past session keys is preserved even when the long term secret, i.e., the password, is disclosed. This property implies that an attacker who knows the password still cannot learn the session key if he only *passively* eavesdrops the key exchange process.

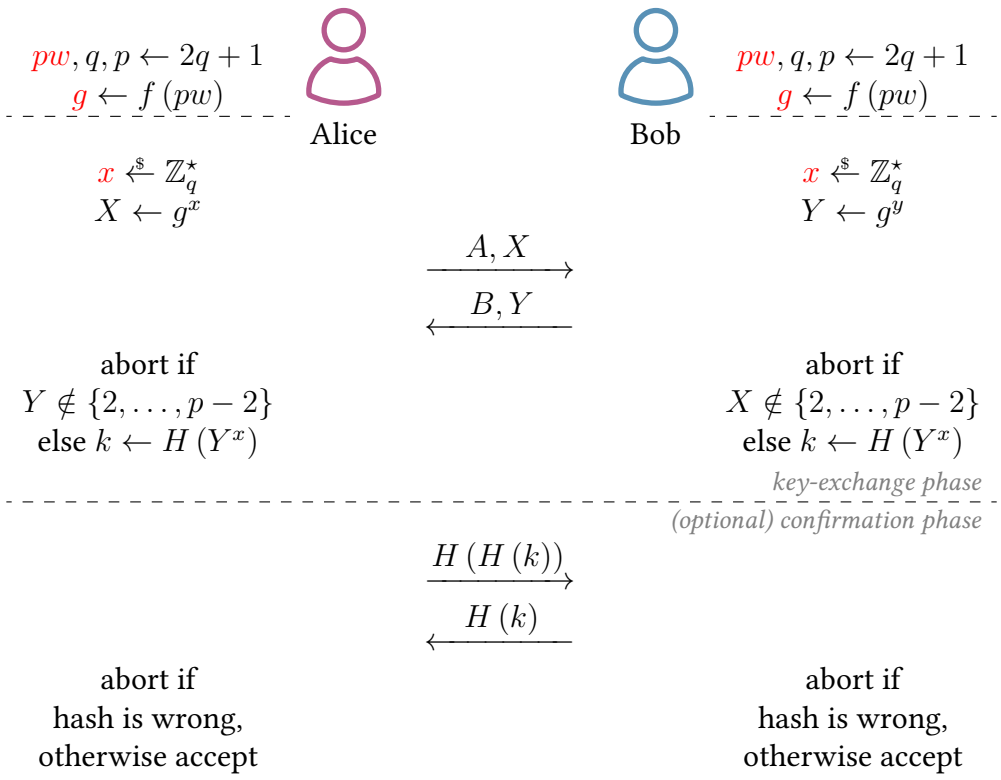
3.1.2 The Original SPEKE

The original specification of the SPEKE protocol in Jablon's 1996 paper [105] is as follows. Participants agreed on a group $\mathbb{G}$ of safe prime order $p = 2q + 1$ where q is also a prime. The SPEKE protocol operates in the subgroup of $\mathbb{G}$ of prime order q where the discrete logarithm problem is assumed to be hard. Two remote parties, Alice and Bob, share a common secret password s from which they apply a function $f(\cdot)$

to compute the group generator: $g = f(s) = s^2 \bmod p$. Unless specified otherwise, all modular operations in the rest of the chapter are performed with respect to the modulus p. We will omit "$\bmod p$" in the notation for simplicity.

The SPEKE protocol runs in two phases: the *key-exchange phase* and the *key-confirmation phase*, as illustrated in Figure 3.1. In the first phase, Alice chooses a secret

Fig. 3.1 Original SPEKE scheme. A and B share the password s and computed $g = s^2 \bmod p$.



value x uniformly at random in $\mathbb{Z}_q^\star = \{1, \ldots, q-1\}$, and sends g^x to Bob. Similarly, Bob chooses a secret value y uniformly at random in $\mathbb{Z}_q^\star$, and sends g^y to Alice. Upon receiving g^y, Alice verifies that its value is between 2 and $p-2$. This is to prevent the small subgroup confinement attack. Subsequently, Alice computes a session key $k = H((g^y)^x) = H(g^{xy})$ where H is a cryptographic hash function (used as a key derivation function here). Similarly Bob verifies g^x belongs to $\{2, \ldots, p-2\}$ and then computes the same session key $k = H((g^x)^y) = H(g^{xy})$. The key-exchange phase is completely symmetric. The symmetry in the design helps simplify the security analysis and reduce the communication rounds especially in a mesh network.

The second phase serves to provide explicit assurance that both parties have actually derived the same session key. This is realized in the original SPEKE paper [105] as follows: one party sends $H(H(k))$ first and the other party replies with $H(k)$ later.

The above key confirmation method has two subtle issues. First, it is ambiguous which party should send $H(H(k))$ first. As we will explain, this ambiguity also carries over to the SPEKE specifications in the ISO/IEC and IEEE standards. Second, from a theoretical perspective, the direct use of the session key in the key confirmation

process renders the session key no longer indistinguishable from random after the key confirmation is finished, hence breaking the session-key indistinguishability requirement in a formal model [36].

In the standards, the key confirmation phase is optional and it is left to the applications to decide whether it is added. With the absence of this phase, key confirmation will have to be deferred to the later secure communication stage where the session key is used to encrypt and decrypt messages (in the authenticated mode) and the decryption will only work if the session keys used at the two sides are equal.

3.1.3 Contribution

The contribution of our work is illustrated in Figure 3.2 and can be summarised as follows.

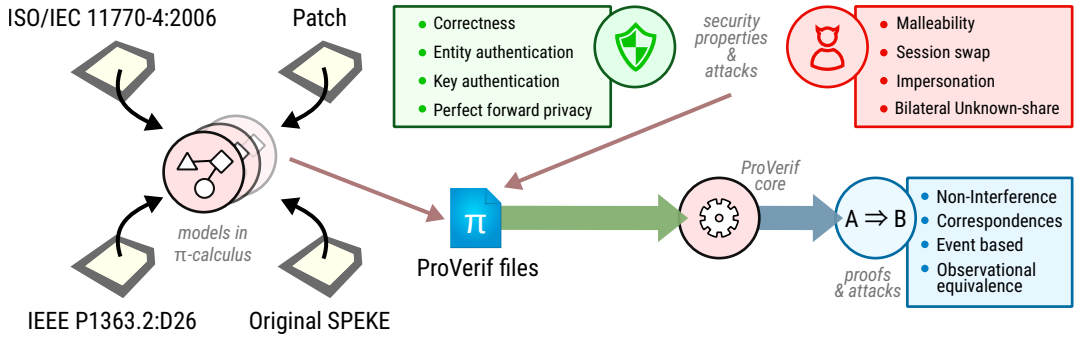


Fig. 3.2 An illustration of our contribution modelling the SPEKE protocol and its security proofs and attacks.

We build a formal model in the Applied Pi Calculus and verify several variants of the SPEKE protocol by using ProVerif. Our model was able to find attacks that showed vulnerabilities to some authentication properties of the protocol; so we propose a patch that verifies the broken properties. The formal model also aided the verification of an improved key confirmation procedure, which is more round-efficient than the one defined in the standards. Finally, we identify an efficiency problem with the key confirmation procedure specified in both the ISO/IEC and IEEE standards and accordingly propose an improved procedure.

The work presented in this chapter extends the earlier conference paper [97] by adding a formal analysis of the patched SPEKE protocol, and details of how the proposed patch was accepted and included into the revision of ISO/IEC 11770-4. The two attacks and the efficiency issues, initially reported in [97], were discussed and acknowledged by the technical committee of ISO/IEC SC 27, Working Group 2. Accordingly, the ISO/IEC 11770-4 standard was revised. The latest revision ISO/IEC 11770-4:2017, incorporating our proposed patch and the improved key confirmation procedure, was formally published in November 2017 [104].

3.1.4 Previous attacks

In [163], Zhang proposed an exponential-equivalence attack on SPEKE. This attack exploits the fact that some passwords are exponentially related. For example, two different passwords s and s' may have the relation that $s' = s^r \bmod p$ where r is an arbitrary integer ($r \neq 1$). By exploiting this relation, an active attacker can rule out two passwords in one go, and in the general case can rule out multiple passwords in one go if they are all exponentially related. This attack is especially problematic when the password is digits-only, e.g., a Personal Identification Numbers (PIN). As a countermeasure, Zhang proposed to hash the password before taking the square operation: in other words, redefining the password mapping function to $f(s) = (H(s))^2 \bmod p$. The hashing of passwords makes it much harder for the attacker to find exponential equivalence among the hashed outputs. Zhang's attack is acknowledged in IEEE P1363.2 [100], which adds a hash function in SPEKE when deriving the base generator from the password.

Tang and Mitchell illustrated three attacks on the SPEKE protocol [157]. The first attack is essentially the same as Zhang's [163]: an active attacker is able to test multiple passwords in one execution of the protocol by exploiting the exponential equivalence of passwords. The authors suggest to hash the identities of the parties along with the password to get the generator, that is $g = H\left(s \| A \| B\right)$ where A and B are identities of two communicating parties. However, this countermeasure has the limitation that it breaks the symmetry of the protocol; instead of allowing the two parties to exchange messages simultaneously in one round, the two parties must first agree whose identity should be put first in the hash, which requires extra communication. The second attack is a *unilateral* Unknown Key-Share (UKS) attack. In this attack, the user is assumed to share the same password with more than one servers[1]. By replaying messages, the attacker may trick the user into believing that he is sharing a key with one server, but in fact he is sharing a key with a different server. To address the attack, they propose to include the server's identity into the computation of g. However, same as before, this countermeasure breaks the symmetry of the original protocol. The last attack they show is a scenario where two sessions are swapped. Here, the two parties run two concurrent sessions, and the attacker swaps the messages between the two sessions. At the end of the protocol, the parties will have shared two session keys, but they may get confused which message belongs to which session. They call this a generic vulnerability, which we call a *sessions swap* attack. To address this problem, they propose to include the "session identifier" into the computation of g, but their paper gives no details on the definition of the "session identifier".

3.1.5 Specification in standards

When SPEKE was included into the IEEE P1363.2 and ISO/IEC 11770-4 standards, the protocol was revised to prevent the exponential-equivalence attack reported

[1]We remark that it is unusual to assume a user shares the same password with multiple server in the security model for PAKE, as a server will be able to trivially impersonate another server. However, in practice, many users do reuse passwords across several accounts.

in [163] and [157]. In the revised protocol, the password is hashed first before computing a secret generator. More specifically, the generator is obtained from $g = (H(s))^2 \bmod p$ instead of $g = s^2 \bmod p$ as in the original 1996 paper.

It is also worth noting that the key confirmation procedure of SPEKE defined in the standards is also different from that in the original SPEKE paper [105]. In IEEE P1363.2 [100] and in ISO/IEC 11770-4:2006 [103], the key confirmation is defined as follows.

$$\begin{array}{ccccc} \text{Alice} & \rightarrow & \text{Bob} & : & H(3\|g^x\|g^y\|g^{yx}\|g) \\ \text{Bob} & \rightarrow & \text{Alice} & : & H(4\|g^x\|g^y\|g^{xy}\|g) \end{array} \tag{3.1}$$

As explicitly stated in the ISO/IEC 11770-4 standard, there is no order in the above two steps. In the same standard, it is also stated that there is no order during the SPEKE exchange phase. We find the two statements contradictory: the fact that g^x comes before g^y in the definition of key confirmation implies there is an order during the key exchange phase.

We would like to highlight that the above issue was carried over from Jablon's original 1996 paper [105], which specifies that "Alice" sends the first confirmation message $H(H(k))$. Given the symmetric nature of the protocol, it is ambiguous which party is "Alice". This ambiguity was unquestioned at the time of standardization and consequently was inherited by the specifications in IEEE P1363.2 and ISO/IEC 11770-4:2006.

We presented the above issue to the ISO/IEC SC 27 technical committee. The issue was acknowledged and rectified in the latest revision ISO/IEC 11770-4:2017. We will explain the details of the change later.

3.2 New attacks

In this section, we review the two new attacks that were reported in [97]: an impersonation attack and a key-malleability attack. We will first explain how the attacks work on the original SPEKE protocol [105] and then explain their applicability to the SPEKE variants defined in the IEEE and ISO/IEC standards [100, 103].

3.2.1 Impersonation attack

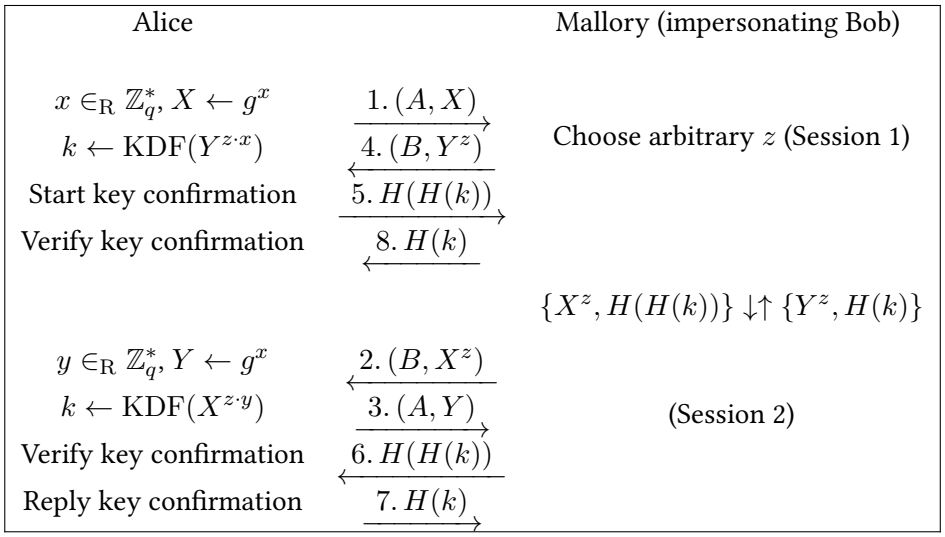
The first attack happens in the setting of parallel sessions: a user is engaged with another user in multiple sessions running in parallel. We illustrate the attack of Mallory who will be able to impersonate the user Bob to Alice, by launching parallel sessions with Alice to make Alice believe she is communicating with Bob, but actually Bob is not involved at all in the communication.

The attack is illustrated in Figure 3.3. Details of each step are explained below.

1. Alice chooses a secret exponent x and computes $X \leftarrow g^x$. She initiates the protocol by sending A, X to the insecure channel.

2. Mallory is in control of the channel and intercepts all the messages to Bob who never receives anything. So, Mallory receives the first message from

Fig. 3.3 Impersonation attack on SPEKE



Alice and generates an exponent z such that $X^z \in \{2, \ldots, p-2\}$[2]. Mallory, impersonating Bob, initiates a parallel SPEKE session with Alice by sending her B, X^z.

3. Alice follows the second session generating an exponent y and computing $Y \leftarrow g^y$. She sends A, Y to the insecure channel.

4. Mallory intercepts the message and raises it to the power of z (with overwhelming probability, Y^z will not be 1 or $p-1$). Then, Mallory sends back to Alice B, Y^z in the first session.

5. At this point, Alice computes the key $k = H\left((Y^z)^x\right) = H\left(g^{xyz}\right)$ for the first session, generates the key confirmation challenge $H(H(k))$, and sends it to Bob.

6. Mallory intercepts the challenge from the first session and relays it to Alice in the second session.

7. Following the protocol, Alice answers the challenge with $H\left(k\right)$.

8. Finally, Mallory intercepts Alice's answer in the second session and replays it in the first session to pass the key confirmation procedure.

At the end of the above attack, Alice authenticates Mallory as "Bob" in both sessions. However, Mallory does not know any secret password and the real "Bob" has never been involved in this key exchange. This indicates a serious flaw in the authentication procedure. We should note that in the above attack, we assume the

[2]When $z = 1$ the work of Mallory reduces to simply relaying Alice's messages to herself in the other session, which may be detected if Alice checks for duplicate of messages.

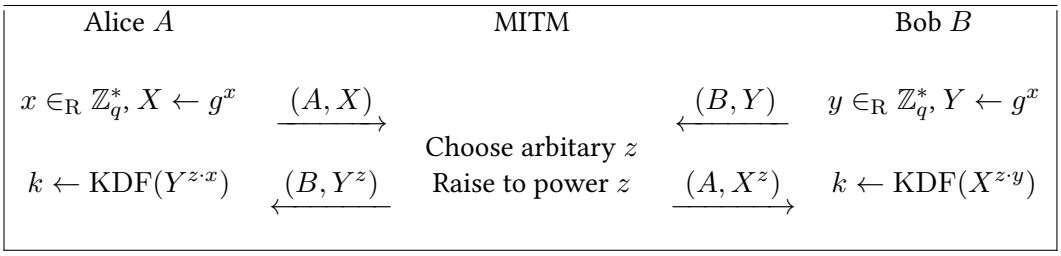
initiator of the session is responsible for sending the first key confirmation message. This is allowed by the protocol since SPEKE specifications in both the IEEE and ISO/IEC standards permit the two parties to start the key confirmation in any order.

This attack can be regarded as a special instance of the Unknown Key-Share (UKS) attack [94]. Alice thinks she is communicating with "Bob", but actually she is communicating with another instance of herself. This confusion of identities in the key establishment can cause problems in some scenarios. For example, using the derived session key k in an authenticated mode, like AES-GCM, Alice may send an encrypted message to Bob: "Please pay Charlie 5 bitcoins". Mallory can intercept this message and (without knowing its content) relay back to Alice in the second session. Since the message is verified to be authentic from "Bob", Alice may follow the instruction (assume Alice is an automated program that follows the protocol). Thus, although Alice's initial intention is to make Bob pay Charlie 5 bitcoins, she ends up paying Charlie instead.

3.2.2 Key-malleability attack

The second attack is a man-in-the-middle attack as shown in Figure 3.4. The attacker chooses an arbitrary z from $\{2, \ldots, q-2\}$, raises the intercepted item to the power of z and passes it on. The parties at the two ends are still able to derive the same session key $k = H(g^{xyz})$, but without being aware that the messages have been modified.

Fig. 3.4 Key-malleability attack on SPEKE



The fact that an attacker is able to manipulate the session key without being detected has significant implications on the theoretical analysis of the protocol. In the original SPEKE paper, the protocol has no security proofs; it is heuristically argued that the security of the session key in SPEKE depends on either the Computational Diffie-Hellman assumption (i.e., an attacker is unable to compute the session key) or the Decisional Diffie-Hellman assumption (i.e., an attacker is unable to distinguish the session key from random). The existence of such a key-malleability attack suggests that a clean reduction to CDH or DDH is not possible. As an example, z can be a result of an arbitrary function $f(\cdot)$ with the incepted inputs, i.e., $z = f(g^x, g^y)$. Because of the correlation of values on the exponent, standard CHD and DDH assumptions are not applicable since they require the secret values on the exponent be *independent*.

This correlation would require additional theoretical efforts to show that the ability to inject such randomness is harmless; in fact, the DDH and CDH require that the secret values are independent.

3.2.3 Discussion on standards

Explicit key confirmation

Recall from Section 3.1.5 that the SPEKE schemes specified in the standards differ from the original SPEKE paper in how the explicit key confirmation is defined. More specifically, the key confirmation procedure in IEEE P1363.2 and ISO/IEC 11770-4 includes additional data in the hash: i.e., key exchange items g^x and g^y. This change does not prevent the impersonation attack; the attacker is still able to relay the key confirmation string in one session to another parallel session to accomplish mutual authentication in both sessions. However, the key-malleability attack no longer works if the key confirmation method in IEEE 1363.2 or ISO/IEC 11770-4 is used. We should emphasize that the key confirmation method in both standards are marked as "optional". Hence, the key-malleability attack is still applicable to the *implicitly authenticated* version of the SPEKE in both standards.

Definition of shared secret

In the earlier conference version of the paper [97], we point out that the definition of the shared secret in ISO/IEC 11770-4:2006 [103] is ambiguous. The shared low-entropy secret, denoted π in that standard document [103], is defined as follows.

> *"A password-based octet string which is generally derived from a password or a hashed password, identifiers for one or more entities, an identifier of a communication session if more than one session might execute concurrently, and optionally includes a salt value and/or other data.*

The above definition seems to include the "identifiers for one or more entities" as part of the shared secret. If the entity identifiers were included, the impersonation attack would not work, but the key-malleability would still work. However, the standard does not provide any formula about π. It is not even clear if one or both entities' identifiers should be included, and if only one identifier is to be included, which one and how. Furthermore, the word "generally" weakens the rigour in the definition and makes it subject to potentially different interpretations.

By comparison, the definition of the shared secret in IEEE P1363.2 (D26) [100] is clearer. It is specified as follows:

> *"A password-based octet string, used for authentication. π is generally derived from a password or a hashed password, and may incorporate a salt value, identifiers for one or more parties, and/or other shared data."*

This definition clearly indicates that the incorporation of "a salt value, identifiers for one or more parties, and/or other shared data" is not mandatory (as indicated by

the use of the word “may”). Based on the definition, it is clear that both attacks are applicable to the SPEKE scheme defined in IEEE P1363.2.

The issue about the ambiguity in the definition was acknowledge by ISO/IEC SC 27 after we first pointed it out in [97], and was rectified accordingly. In the latest revision in ISO/IEC 11770-4:2017, the definition of the shared secret has been revised to follow the same as in IEEE P1363.2 (D26) [100]. In this revision, the two reported attacks are addressed by making technical changes to the SPEKE specification, as we will explain in the next section.

3.3 Patched SPEKE

There are several reasons to explain the cause of the two attacks. First, there is no reliable method in SPEKE to prevent a sent message being relayed back to the sender. Second, there is no mechanism in the protocol to verify the integrity of the message, i.e., whether they have been altered during the transit. Third, no user identifiers are included in the key exchange process. It may be argued that all these issues can be addressed by using a Zero Knowledge Proof (ZKP) (as done in [96]). However, in SPEKE, the generator is a secret, which makes it incompatible with any existing ZKP construction. Since the use of ZKP is impossible in SPEKE, the attacks were addressed in a different way.

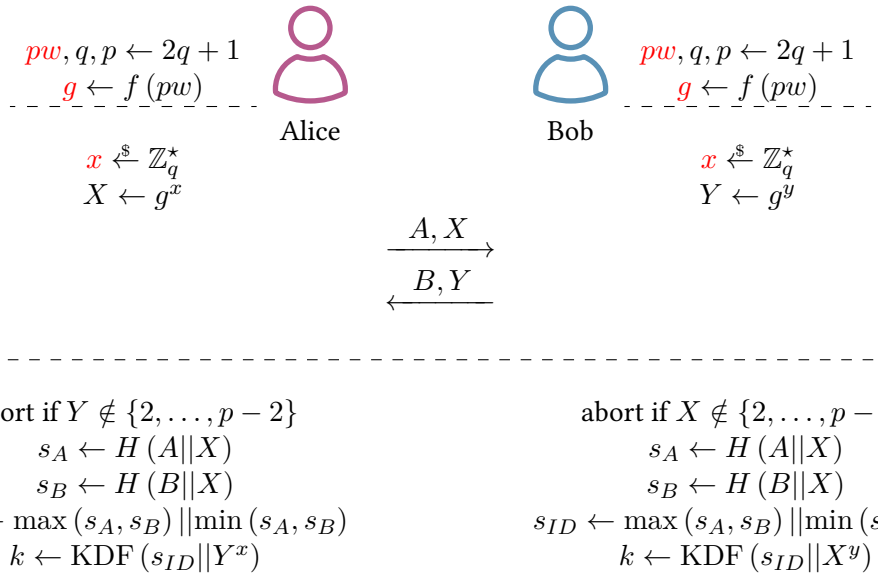
Our proposed patch is to redefine the session key computation. Assume Alice sends g^x and Bob sends g^y. The session key computation is defined below.

$$\begin{aligned} s_A &= H(A \| g^x) \\ s_B &= H(B \| g^y) \\ s_{ID} &= \max(s_A, s_B) \| \min(s_A, s_B) \\ k &= \mathrm{KDF}(s_{ID} \| g^{xy}) \end{aligned} \tag{3.2}$$

When the two users are engaged in multiple concurrent sessions, they need to ensure the identifiers are unique between these sessions. As an example, assume Alice and Bob launch several concurrent sessions. They may use “Alice” and “Bob” in the first session. When launching a second concurrent session, they should add an extension to make the entity identifier unique – for example, the entity identifiers may become “Alice (2)” and “Bob (2)” respectively in the second session, and so on. The use of the extension is to make the entity identifier distinguishable among multiple sessions running in parallel.

The new definition of the session-key computation function in Eq. 3.2 prevents both the impersonation and key-malleability attacks (as well as the *session swap* attack reported in [157]), which we will formally prove in the next section. The key confirmation remains “optional" as it is currently defined in the standards. Furthermore, this patch preserves the optimal one-round efficiency of the original SPEKE protocol.

Fig. 3.5 Patched SPEKE (included in ISO/IEC 11770-4:2017 [104]).



An alternative patch, suggested in the earlier conference paper [97], is to refine the session key computation as follows.

$$\begin{aligned} M &= H(\min(A,B)\|\max(A,B)) \\ N &= H(\min(g^x,g^y)\|\max(g^x,g^y)) \\ k &= \mathrm{KDF}(M,N,g^{xy}) \end{aligned} \tag{3.3}$$

As we will formally analyse in Section 3.4, the above solution also prevents the two attacks. However, the advantage of the solution in Eq. 3.2 is that the hash output has a fixed bit length, which makes it easier to implement the max and min functions. The final patch, which has been included into the latest revision of ISO/IEC 11770-4 published in 2017, is summarized in Figure 3.5.

3.3.1 Improved key confirmation

As highlighted in Section 3.1.5, neither of the key confirmation procedures defined in IEEE P1363.2 (D26) and ISO/IEC 11770-4 (2006) is symmetric. In both standards, they state that there is "no special ordering" of the key confirmation message. This implies that the messages can be sent simultaneously within one round. But in fact, these procedures require two rounds instead of one, because the second message depends on the first. This issue also applies to the key confirmation method in Jablon's original 1996 paper [105]. If both parties attempt to send the first message at the same time without an agreed order, they cannot tell if the message that they receive is a genuine challenge or a replayed message, and consequently enter a deadlock.

To address the above issue, we propose an improved key confirmation method which preserves the symmetry of the protocol and hence allows the key confirmation

to be completed within one round. It works as follows.

$$\begin{array}{rcrcl} \text{Alice} & \rightarrow & \text{Bob} & : & H(A\|B\|g^x\|g^y\|g^{xy}\|g) \\ \text{Bob} & \rightarrow & \text{Alice} & : & H(B\|A\|g^y\|g^x\|g^{xy}\|g) \end{array} \tag{3.4}$$

An alternative solution, proposed in our earlier paper [97], is based on NIST SP 800-56A Revision 1 [19]. It works as follows.

$$\begin{array}{rcl} \text{Alice} \rightarrow \text{Bob} & : & \text{MAC}(k_c, \text{“KC\_1\_U”}\|A\|B\|g^x\|g^y) \\ \text{Bob} \rightarrow \text{Alice} & : & \text{MAC}(k_c, \text{“KC\_1\_U”}, \|B\|A\|g^y\|g^x) \end{array}$$

In the above method, MAC is a message authenticated code (MAC) algorithm, the string “KC_1_U” refers to unilateral key confirmation, and k_c is a MAC key. To allow the session key to remain indistinguishable from random even after the key confirmation phase, k_c should be derived differently from the session key, e.g., by adding a specific parameter to the key derivation function say $k_c = \text{KDF}(g^{xy}, \text{“KC”})$. There is no dependence between the two flows, so Alice and Bob can send messages in one round. During the revision of ISO/IEC 11770-4, the hash based key confirmation method in Eq. 3.4 was preferred and was included into the latest standard since it requires minimum changes in the standard.

3.4 Formal analysis

We formally model the following variants of the SPEKE protocol in the Applied Pi Calculus [142]: the original Jablon's protocol [105], the ones in IEEE P1363.2:D26 [100] and ISO/IEC 11770-4:2006 [103], the earlier patch proposed by Hao and Shahandashti in 2014 [97], and the final patch described in our paper [95] and included into ISO/IEC 11770-4:2017 [104], each in two modes:

- without explicit key confirmation,
- with explicit key confirmation as described in the respective documents.

It is worth noting that a meaningful key exchange process should always be completed with some form of key confirmation, let it be explicit or implicit. The explicit key confirmation is realized by executing the explicit key confirmation procedure, which requires extra rounds of communication. But the explicit key confirmation procedure is optional [100, 103]: without it, the key confirmation is deferred to the secure communication stage, and this is called *implicit* key confirmation [153]. However, the exact mechanisms for *implicit* key confirmation are not specified in [105, 100, 103], which makes it difficult to model SPEKE with implicit key confirmation. To address this issue, we assume the implicit key confirmation is realized in the secure communication stage by prepending the first encrypted message with an explicit key confirmation string as defined in the respective explicit key confirmation procedure. Thus, our formal model treats SPEKE with implicit and explicit key confirmations as essentially the same with the only difference being that the latter requires additional rounds of communication.

In the model, we formally specify the following:

The two parties. Following the formalism illustrated in Section 2.4, the original SPEKE protocol illustrated in Figures 3.1 is constituted by two processes, one for the Initiator, P_I, and one for the Responder, P_R. All variants of the SPEKE protocol involve two parties: the Initiator I and the Responder R. They are modelled as two processes P_I and P_R. We use the initiator and the responder for the convenience of naming in our model. Essentially we assume that one party initiates the protocol by sending data in the first flow, and the other party responds by sending data in the second flow. Thus a one-round protocol is implemented in two flows. This does not change the security analysis of the protocol. Below we give the "vanilla" specification of the protocol. In the "vanilla" specification, we abstract out key reconstruction by a function symbol kdf, and the confirmation messages sent by the Initiator and the Responder are abstracted by the symbols kcf_I and kcf_R respectively. The actual specification of each variant has its own definitions of kdf, kcf_I and kcf_R to capture the differences between the variants.

Fig. 3.6 The processes for the Initiator, P_I, and the Responder, P_R. In the above specification, the notation $= X$ means abort if the incoming value is not X.

$$
\begin{array}{l}
1 \quad P_I \leftarrow \mathrm{in}\,(c,(I,R))\,; \\
2 \quad\quad \mathrm{get}\ t\,(=I,=R,g)\ \mathrm{in} \\
3 \quad\quad \nu x.\mathrm{let}\ X = g^x\ \mathrm{in} \\
4 \quad\quad \mathrm{out}\,(c,(I,X))\,; \\
5 \quad\quad \mathrm{in}\,(c,(=R,Y))\,; \\
6 \quad\quad \mathrm{let}\ k = kdf\ \mathrm{in} \\
7 \quad\quad \mathrm{out}\,(c,kcf_I)\,; \\
8 \quad\quad \mathrm{in}\,(c,=kcf_R)\,; \\
9 \quad\quad \mathrm{out}\,(c,\mathrm{enc}\,(k,m))\,;
\end{array}
\qquad
\begin{array}{l}
1 \quad P_R \leftarrow \mathrm{in}\,(c,(I,R))\,; \\
2 \quad\quad \mathrm{get}\ t\,(=R,=I,g)\ \mathrm{in} \\
3 \quad\quad \nu y.\mathrm{let}\ Y = g^y\ \mathrm{in} \\
4 \quad\quad \mathrm{out}\,(c,(R,Y))\,; \\
5 \quad\quad \mathrm{in}\,(c,(=I,X))\,; \\
6 \quad\quad \mathrm{let}\ k = kdf\ \mathrm{in} \\
7 \quad\quad \mathrm{in}\,(c,=kcf_I)\,; \\
8 \quad\quad \mathrm{out}\,(c,kcf_R)\,; \\
9 \quad\quad \mathrm{out}\,(c,\mathrm{enc}\,(k,m))\,;
\end{array}
$$

As can be easily seen in Figure 3.6, the code inside the boxes is the part modelling the protocol scheme depicted in Figure 3.1, where the key reconstruction part is abstracted by the function symbol kdf, and the confirmation messages sent by the Initiator and the Responder are abstracted by the symbols kcf_I and kcf_R respectively, where we omit their arguments for simplicity. We highlight the symmetric nature of the protocol letting both processes to write to the channel simultaneously.

The other lines (outside the box) in Figure 3.6 serve to model the behaviour of the protocol and to verify some security properties[3]. In particular, the first line is to let the processes know the identities involved in the protocol; they read them from the channel c at the very beginning. The second line checks whether the password table

[3]It can be seen as a trick to make ProVerif happy and simplify the modelling of the secrecy of the key.

contains a suitable password to communicate to the other party; otherwise, they abort. The last line is useful to verify the secrecy of the shared key k through the privacy of the message m, and the perfect forward secrecy. The details are deferred to Section 3.4.2 where we discuss the security properties.

The pre-shared password. A table t of passwords is filled with all secret group generators that would be calculated from the passwords, i.e., $g \in \mathcal{G}$ is the secret group generator for A and B. From the point of view of the symbolic protocol design, sharing a password and then computing the generator is the same as having directly shared the secret generator.

The main process. Informally, the main process P is an infinite repetition of the parallel execution of the Initiator's process P_I and the Responder's process P_R. Due to the symmetric nature of the SPEKE protocol, the naive implementation of the main process brings *false* attacks where the Initiator speaks to itself. To avoid this issue, we must explicitly support the session within the two parties. However, the session s is not private information, and we disclose it to the attacker by outputting it to the insecure channel c, i.e. $\mathrm{out}\,(c, s)$. At this point, we have the infinite repetition of the following process: $!\,(\nu s.\mathrm{out}\,(c, s)\,;(P_I|P_R))$. The two parties would never engage the protocol if they do not share the password. For this reason, we have an environment process P_P which is in charge of inserting shared passwords into a table that can be accessed by P_I and P_R, but not the attacker. In order to record events and verify correspondences, we also have a process P_A, which records the agreements between the parties through events.

Finally, the main process P that the tool checks has the following structure:

$$P \leftarrow (P_P|\,(!\,(\nu s.\mathrm{out}\,(c, s)\,;(P_I|P_R)))\,|!P_A)\,.$$

The process P_A collects information from two tables, one filled in by the Initiator and the other by the Responder. We emphasise that the protocol can be initiated by either of them and the two tables are put together recording a single event. For security properties that do not require tables to record events, the process will be simply **0**; otherwise, depending on the property to prove, the events e_S end e_C can be recorded, where e_S means that the involved parties in the protocol agree with the participants, the session, and the reconstructed session key at the end of the protocol, and e_C means that the involved parties in the protocol agree with the participants, the session, the secret group, the secret nonce, and the reconstructed session key at the end of the protocol.

Modelling attacks and security properties

The reasoning engine of ProVerif will execute a *main* process and record traces of execution. The traces are potentially infinite, as the amount of concurrent executions of the protocol is unbounded; however, the reasoning core contains built-in theorems that detect some special structure in the trace expansion that are able to provide final statements over infinite traces without the need to expand them. Attacks and security properties are both statements that relate to the infinite traces of execution. A security property is a statement over those traces that capture a desirable behaviour

of the protocol. The difference with attacks is that an attack depicts a situation that is not desirable, so the *resilience* against the attack can be seen as the negation of the relation. We modelled some attacks as either direct attacks or *resilience* to those attacks, as sometimes a negation to a statement can yield to a more complex formula, so we found convenient sometimes to interpret the final result manually. So, the Sections 3.4.1 and 3.4.2 show which statements we modelled to capture the corresponding attacks or security properties.

3.4.1 Attacks

The attacks are modelled as follows and results are shown in Table 3.1.

Table 3.1 Summary of results of attacks in ProVerif.

Variants	IMP	SS	UKS	MAL
Jablon 1996 [105]	×	×	×	×
IEEE P1363.2:D26 [100]	×	×	×	✓
ISO/IEC 11770-4:2006 [103]	×	×	×	✓
ISO/IEC 11770-4:2017	✓	✓	✓	✓

The results are grouped by variants with and without key confirmation phase (KC).
Legend. Impersonation (IMP), Sessions Swap (SS), bilateral Unknown Key-Share (UKS), and Malleability (MAL). **Outcomes**: (✓) - no attacks possible, (×) - attacks found.

An important comment about the attacks is that they strictly relate to security properties: for example, if strong entity authentication (described in Section 3.4.2) is met, then impersonation and session swap attacks would be automatically ruled out. However if only weak entity authentication is met, still impersonation should not be possible, but a session swap attack (among the same authenticated entities) may still be viable.

Bilateral UKS

Informally, a successful bilateral UKS attack makes two honest parties I and R believe that they share k with some other party [62]. To capture this attack, we use the following correspondence:

$$\forall h_1,h_2,h_1',h_2' \in \mathcal{H}, s,s' \in \mathcal{S}, k \in \mathcal{K}.$$
$$e_S\left(h_1,h_2,s,k,h_1',h_2',s',k\right) \Rightarrow h_1 = h_1' \wedge h_2 = h_2'$$

If an initiator and a responder recorded the same key, then it must be that they agree on the entities. If we required that they should agree on the session too, then we could put $s = s'$ in logical AND with the two equivalences. On the contrary, if we wanted to force the tool to show bilateral UKS attacks in the same session, we could state $s = s'$ as a premise.

Impersonation attack

The impersonation attack is a problem that generally affects SPEKE protocols and an instance of such an attack has been shown in Section 3.2.1. To formalise this attack, we build a model in which there exists only one honest party and the attacker. In this case, if the honest party ever shares a key with another party, then the other party must be the attacker, and the attacker must impersonate another honest party in order to run the protocol up to this point. In fact, all SPEKE variants without key confirmation phase are vulnerable to this attack.

To verify, we can check for every honest party, session and key, the event of authenticating the other party is not recorded in any trace (i.e. the adversary cannot establish a shared key with the honest party). Formally, we check the following property:

$$\forall h_1, h_2 \in \mathcal{H}, s \in \mathcal{S}, k \in \mathcal{K}. \\ \neg \left(e_{RI}\left(h_1, h_2, s, k\right) \vee e_{IR}\left(h_1, h_2, s, k\right)\right).$$

When such property is verified, any impersonation cannot be carried out.

Sessions swap

A man-in-the-middle is able to perform the sessions swap attack if it can let a honest party in some session s share a key with another honest party in some other concurrent session s' and vice versa. This attack occurs in a key-exchange protocol where the key does not depend on the session. Formally, we say that for every two parties and for every key, the presence of an agreement on the Initiator, Responder, and session key must imply the equivalence of the sessions.

$$\forall h_1, h_2 \in \mathcal{H}, s, s' \in \mathcal{S}, k \in \mathcal{K}. \\ e_S\left(h_1, h_2, s, k, h_1, h_2, s', k\right) \Rightarrow s = s'.$$

Malleability

The malleability of the session key is an attack that affects many variants of the SPEKE protocols, and it was described in Section 3.2.2. The attacker can *simply* raise to the power of the same exponent z the two messages exchanged in the first phase of the protocol before delivering them to the intended parties. This behaviour allows the attacker to change the value of the exchanged key from g^{xy} to g^{xyz}, where x and y are the secret fresh exponents generated by respectively the initiator and the responder. Capturing malleability in ProVerif is not directly supported, but cases in which it can be done are not excluded a priori. We modelled malleability similarly to how the manual suggests for modelling properties requiring two commutative exponents for the Diffie-Hellman based protocols. [4] Capturing the malleability attack in ProVerif requires more efforts than other attacks, because it is based on an *extra*

[4] When we tried to implement a further level of group exponentiation equality, the computation required weeks instead of minutes even for the easiest verification in a modern workstation.

level of group exponentiation equality (three commutative exponents). This results in a larger search space when the reasoning engine checks the property, and ProVerif slows down considerably (taking minutes instead of milliseconds to verify the non-malleability property on a 3.2 GHz computer with 64 GB RAM running Linux). To detect malleability, we require the two honest parties to write into a table some values they agree with, plus their secret fresh exponents and the secret generator (the password). This way, when checking for correspondence, we can check whether the key is indeed what is expected with regard to the private inputs of the parties.

Formally, for every pair of parties, session, generator, two exponents and key, where the parties agree on the identities, the session, the generator and the key (they cannot agree on the other party's secret), then the key they agree on is computed equivalently to the formula provided by the protocol.

$$\begin{aligned}&\forall h_1, h_2 \in \mathcal{H}, x, y \in \mathbb{Z}_q^\star, g \in \mathbb{Z}_p^\star, s \in \mathcal{S}, k \in \mathcal{K}.\\ &\quad e_C\left(h_1, h_2, s, h, x, k, h_1, h_2, s, g, y, k\right) \Rightarrow\\ &\quad k = kdf\left(a, b, g^x, g^y, g^{xy}, s\right) \vee kdf\left(b, a, g^y, g^x, g^{xy}, s\right).\end{aligned}$$

Note the key k may have two different values depending on in the protocol who is the initiator and who is the responder.

We remark that the kdf is expected to have the additional property of being commutative in the first two pairs of arguments, emphasising the symmetric nature of the protocol.

3.4.2 Security properties

The security properties are modelled as follows and results are shown in Table 3.2.

Table 3.2 Summary of results on formal verification of security properties in ProVerif.

Variants	IKA	EKA	WA	SA	PFS
Jablon 1996 [105]	✓	×	×	×	✓
IEEE P1363.2:D26 [100]	✓	×	×	×	✓
ISO/IEC 11770-4:2006 [103]	✓	×	×	×	✓
ISO/IEC 11770-4:2017	✓	✓	✓	✓	✓

The results are grouped by variants with and without key confirmation phase (KC).
Legend. Implicit Key Authentication (IKA), Explicit Key Authentication (EKA), Weak Entity Authentication (WA), Strong Entity Authentication (SA), Perfect Forward Secrecy (PFS). **Outcomes**: (✓) - verified, (×) - attacks found.

Correctness

This property checks whether the protocol gives authentication and key distribution in presence of honest parties [161]. With respect to *executability*, which is a very similar property that simply tests whether the protocol reaches the end, the property of correctness additionally checks if at the end of the execution, the two parties

indeed hold what they were supposed to, in our case the correct key. Even though correctness is generally the easiest to prove, it should not be neglected when formally modelling a protocol, in order to avoid either logical or typographic errors[5]. To check the correctness of the models, we need to reconstruct the session key kdf. Its implementation depends on the SPEKE variants.

Formally, for all the sessions and nonce exponents, we require that there exists at least a trace in which the event collecting private and shared values of the participants is recorded and is such that the two honest participants agree on their identities, the password, and the session key with the right formula.

$$\begin{aligned} \forall\, s \in \mathcal{S} \quad & x, y \in \mathbb{Z}_q^\star, g \in \mathbb{Z}_p^\star. \\ & e_C(A, B, s, g, x, kdf\ (A, B, g^x, g^y, g^{xy}, s)\,, \\ & \quad A, B, s, g, y, kdf\ (A, B, g^x, g^y, g^{xy}, s)) \end{aligned}$$

where A and B are honest parties and g is the generator calculated from the shared password. If such an event is raised, then there exists a run of the protocol in which the two parties have authenticated each other and they have correctly computed the session key.

Secrecy of the pre-shared password

Secrecy of the pre-password can be modelled in two ways, observational equivalence or by inspection of the attacker's knowledge whether it can infer the secret with combinations of elements in the trace of execution. We chose the former, as it covers a stronger notion of security[6]. Formally, if we call π_g the process describing the protocol where two honest parties A and B share the password g, and $\pi_{g'}$ the same protocol but with g' instead of g, then the observational equivalence $\pi_g \approx \pi_{g'}$ describes the property that any attacker cannot distinguish between the two runs of the protocol with probability (non-negligibly) better than a blind guess, and therefore no extra information about the secret password can be gained.

Implicit key authentication

Implicit key authentication is verified when only the two participants can reconstruct the session key. This concept is modelled by using the key to encrypt a secret message with deterministic encryption; event based properties (similar to the events we used for explicit authentication) could have been used as well but they would have not add any benefit. We then check for observational equivalence of two runs of the processes P_I and P_R where in the last line (Figure 3.6) the message encrypted is provided by a choice, $\mathrm{out}\,(c, \mathrm{enc}\,(k, [m, m']))$. Similar to how we determine the secrecy of the password, if we call π_m the process describing the protocol where two honest parties A and B encrypt m, and $\pi_{m'}$ the same protocol but with $m' \neq m$, then the observational equivalence $\pi_m \approx \pi_{m'}$ is verified. If the observational equivalence

[5]Executability may not fully check this as the last algorithm may still conceal undetected typos, that, depending on how the key exchange protocol is modelled, it may yield to incorrect key.

[6]Observational equivalence may not terminate in ProVerif, but luckily it did for our model.

holds and therefore the message m remains secret, it trivially follows that the shared key is at least as secret as m. In fact, the decryption function is public, and the reconstruction of the key will irredeemably compromise the secrecy of m.

Explicit key authentication

Explicit key authentication is verified when only the two participants can reconstruct the session key, and they actually do. It is therefore defined as implicit key authentication *and* an agreement on the computed key for the same session. Formally,

$$\begin{aligned}&\forall h_1, h_2 \in \mathcal{H}, s \in \mathcal{S}, k, k' \in \mathcal{K}.\\ &\qquad e_S\left(h_1, h_2, s, k, h_1, h_2, s, k'\right) \Rightarrow k = k'\end{aligned}$$

In other words, in a trace of execution the presence of the event e_S where the first and fifth arguments being equal (agreement on the initiator), the second and the sixth being equal (agreement on the responder), and the third and the seventh being equal (agreement on the session) implies that the fourth and the eighth are equal (equivalence of the reconstructed key). When this property is true, a protocol completed between two authenticated parties in the same session guarantees that the parties agree on the session key. This property, along with the implicit key authentication, gives explicit key authentication.

Weak and strong entity authentication

Weak entity authentication guarantees that two parties are indeed speaking to each other. Strong entity authentication requires agreement on other values than the mere entities. These values are supposed not to be injected, produced or inferred by an attacker. Those two properties share similarities in their formality. The events involved are 1) e_I to record that the initiator I believes that it has started a protocol with the responder R; 2) e_R to record that R believes that it has started a protocol with I; 3) e_{IR} to record that I believes that it speaks to R at the end of the protocol, and 4) e_{RI} to record that R believes that it speaks to I at the end of the protocol. The first and the third are recorded by the honest initiator, while the the second and the fourth by the honest responder. Mutual *weak* authentication is provided by the two following symmetric correspondences, one for each honest party:

$$\begin{aligned}&\forall h_1, h_2 \in \mathcal{H}.\ e_{IR}\left(h_1, h_2\right) \Rightarrow e_R\left(h_1, h_2\right)\\ &\forall h_1, h_2 \in \mathcal{H}.\ e_{RI}\left(h_1, h_2\right) \Rightarrow e_I\left(h_1, h_2\right)\end{aligned}$$

And mutual strong authentication by the following:

$$\begin{aligned}&\forall h_1, h_2 \in \mathcal{H}, s \in \mathcal{S}, k \in \mathcal{K}.\\ &\qquad e_{IR}\left(h_1, h_2, s, k\right) \Rightarrow e_R\left(h_1, h_2, s, k\right)\\ &\forall h_1, h_2 \in \mathcal{H}, s \in \mathcal{S}, k \in \mathcal{K}.\\ &\qquad e_{RI}\left(h_1, h_2, s, k\right) \Rightarrow e_I\left(h_1, h_2, s, k\right)\end{aligned}$$

where they agree also on the session and the exchanged session key. Agreeing on the session will prevent any replay attack from other sessions, even concurrent, while agreeing on the key will guarantee that no attacker can let two authenticated parties not to share the same key. However, a key-malleability attack is still possible even if the protocol can achieve strong entity authentication.

Perfect forward secrecy

Usually, key exchange protocols verify (or claim) the perfect forward secrecy (PFS) property. For the password authenticated key exchange protocols, this property means that if passwords are compromised, the *past* session keys derived from such passwords still remain secret. Hence, an adversary can only keep a record of past communication which has not been compromised. We can reformulate this concept as a *passive* adversary whom is given the password and eavesdrops (unbounded number of) executions of the protocol trying to reconstruct any of the session keys. In practice, to verify this property we disclose the secret generator g to the attacker, $\mathrm{out}\,(c, g)$, then we query the non-interference property on the encrypted message. Since the passive attacker can compute any decryption, the non-interference property captures the perfect forward secrecy, i.e., if the encrypted message cannot be reconstructed, it must be that any session key cannot be reconstructed either.

3.5 Summary of results

The ProVerif scripts that we created to model and verify the protocols are available at GitHub[7]. There are in total 54 scripts related to our formal analysis, each for a different variant and a property[8]. ProVerif will give one of the following four responses: (i) the property is true, (ii) the property is false, (iii) the property cannot be proved, and (iv) non-termination. A property cannot be proved when an attack at the Horn clauses has been found but no attacks at the Pi Calculus level of abstraction can be reconstructed. This happens because the translation from the Pi Calculus to the Horn clauses is such that what can be proved at the level of Horn clauses implies a truth value at the Pi Calculus level; however if an attack to a property (a false) is found at the Horn clauses level, it might be a false attack due to limits in the reasoning core: hence, the need for reconstructing the attack at the Pi Calculus level to detect (possibly) false attacks.

We formally modelled many variants of the SPEKE protocol in ProVerif. Both variants with and without key confirmation (KC) are compared in Table 3.3.

The original SPEKE with KC was proposed in two flavours. In the first, they engage a challenge/response communication, while in the second, the two parties send a hash to each other. Both of them show the same vulnerabilities. Similarly, the IETF I-D v02 is the only variant in which we modelled the B-SPEKE and the W-SPEKE. They are different ways of computing the key, but they address security properties about offline dictionary attacks or Denning-Sacco attacks. Since we do not model

[7] https://github.com/nitrogl/speke-verification

[8] In ProVerif some security properties are incompatible and cannot lie in the same file.

Table 3.3 Summary of results on efficiency and formal verification in ProVerif

Variant	RND/E	IKA	EKA	WA	SA	PFS	IMP	SS	UKS	MAL
Jablon 1996 [105]	1/3	✓	×	×	×	✓	×	×	×	×
IEEE P1363.2:D26 [100]	1/3	✓	×	×	×	✓	×	×	×	✓
ISO/IEC 11770-4:2006 [103]	1/3	✓	×	×	×	✓	×	×	×	✓
ISO/IEC 11770-4:2017	1/2	✓	✓	✓	✓	✓	✓	✓	✓	✓

The results are grouped by variants with and without key confirmation phase (KC).
Legend. **Round efficiency**: without explicit key confirmation (RND), with explicit key confirmation (E). **Security properties**: Implicit Key Authentication (IKA), Explicit Key Authentication (EKA), Weak Entity Authentication (WA), Strong Entity Authentication (SA), Perfect Forward Secrecy (PFS). **Attacks**: Impersonation resilience (IMP), Sessions Swap resilience (SS), bilateral Unknown Key-Share resilience (UKS), and Malleability resilience (MAL). **Outcomes**: (✓) - verified/no attacks, (×) - attacks found.

such properties, their formal verification gives the same results, and we included them for completeness. They must be grouped by this first difference, because many security properties are expected either not to be verified or not to make any sense, i.e. Explicit Key Authentication (EKA) is never expected in the one-round variants of the protocol since there is no explicit confirmation of the exchanged key. We decided not to include our model of the patch proposed by Tang and Mitchell [157] because it was not very clearly proposed and for what we tried to model, they cannot reach strong authentication nor verify sessions swap resilience.

The results are summarised in Table 3.3. The proposed patch (as well as the patch in [97]) improves the round efficiency over the previous SPEKE variants [105, 100, 103] by allowing the explicit key confirmation steps to be completed within one round. As a result, it requires only 2 rounds to finish the key exchange with explicit key confirmation as opposed to 3 rounds previously. All variants have the Implicit Key Authentication (IKA) property, confirming that the session key will not be learned by the attacker, and that the attacker cannot get confidential information by eavesdropping. This does not contradict the impersonation attack shown in Section 3.2, since that attack works without the adversary learning the session key. However, that attack demonstrates that the adversary is able to manipulate the two parallel sessions to make them generate *identical* session keys. Consequently, the adversary is able to pass the explicit key confirmation by replaying messages. This is confirmed by our formal analysis that the original SPEKE [105], and the SPEKE in standards [100, 103] do not fulfil the explicit key authentication property. Also, the existence of the impersonation attack shows that these variants do not fulfil the weak/strong entity authentication which concerns assuring the identities of the entities involved in the key exchange protocol. The proposed patch prevents the Session Swap attack (SS), the UKS attack, and the Malleability (MAL) attack by making the session key depend on the session, the identities, and the transcript of the key exchange process. We emphasise that these security properties are verified *before* any key confirmation either implicit or explicit. To guarantee that the participants are mutually authenticated, the key confirmation becomes necessary. Such key confirmation must include all of the key points above, i.e., session, identities, and a

transcript of the key exchange messages, so avoiding the above mentioned attacks. Including only the identities allows to verify *weak* entity authentication only.

Our formal analysis using ProVerif confirms that our proposed patch prevents the two attacks as identified earlier. However, this analysis does not constitute a complete proof of security for SPEKE, as one might expect from formal authenticated key exchange models [36, 35, 110, 88, 8]. In particular, we have not proved that SPEKE is resistant against off-line dictionary attacks based on standard security assumptions such as DDH or CDH. We highlight that the original SPEKE was designed without a security proof. Retrospective efforts to prove the security of a protocol based on standard number theoretical assumptions may turn out to be very hard if not impossible. We leave further analysis of SPEKE to future work.

3.6 Conclusions

The SPEKE protocol was firstly proposed by Jablon over two decades ago. Since then, it has been adopted by international standards, and built into smart phones and other products. We identified two weaknesses in the standardized SPEKE specification, which affect all implementations that follow the IEEE 1362.3 and ISO/IEC standards. Accordingly we proposed a patched SPEKE to address the identified issues. We formally modelled the discovered attacks against SPEKE and proved that the proposed patch was not affected by these attacks, but this does not rule out other attacks not covered by our model. In addition, we contributed to improve the round efficiency of the protocol in the key confirmation phrase. Our proposed patch and the improved key confirmation procedure have been included into the latest revision ISO/IEC 11770-4 published in July 2017. However, the SPEKE specification in IEEE P1363.2 (which is currently not maintained) remains unfixed.

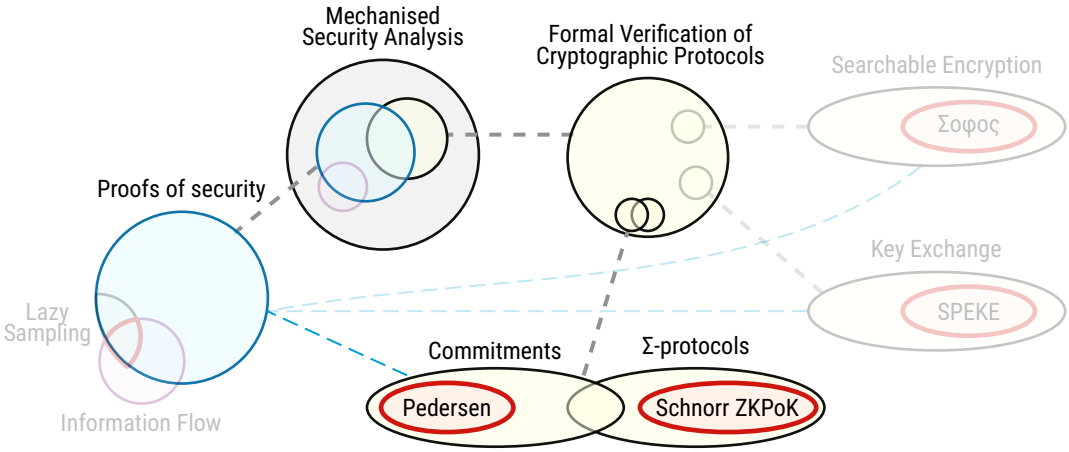
The problems in SPEKE discussed in this chapter have evaded 20 years cryptanalysis (informal and formal) by the security and standardization communities. Reasons can be manifold. One reason can be that its popularity was not comparable to other protocols, despite of its adoption in several commercial applications: for example, the secure messaging on Blackberry phones [2] and Entrust's TruePass end-to-end web products [1], that became obsolete. Another reason can be that it relies on pre-shared password, which requires additional care when modelling, so that tools needed to offer more capabilities and only after decades of research we can enjoy more mature verification tools. Another reason can be that it was previously modelled in another tool, AVISPA [160], which did not find any attacks; this might have discouraged researchers to model SPEKE capturing more details, as we did, and see if they could find indeed problems.

The initial discovery of the two attacks on SPEKE was down to manual analysis, which was later formally verified by applying the ProVerif tool. Importantly, the formal analysis shows that those (an many more) attacks cannot affect the patch we proposed, strongly supporting its security claims. The mechanised proofs that we produce are not only helpful for proving security properties of similar protocols, but also as a remainder that the same problems may happen in other protocols. The lesson we learned is that traditional human cryptanalysis can be significantly aided by

using modern automated proof techniques in at least two aspects: the first is that the machines can be exhaustive when inspecting all (modelled) scenarios and therefore can find attacks where human cryptanalysis is limited, the second is that formal and mechanised analysis strengthens security arguments with additional evidence and improved reproducibility. This will result in improving security protocols, especially those that have been included in international standards and are more likely to be adopted for compliance.

Chapter 4

Commitment and Sigma protocols in the computational model



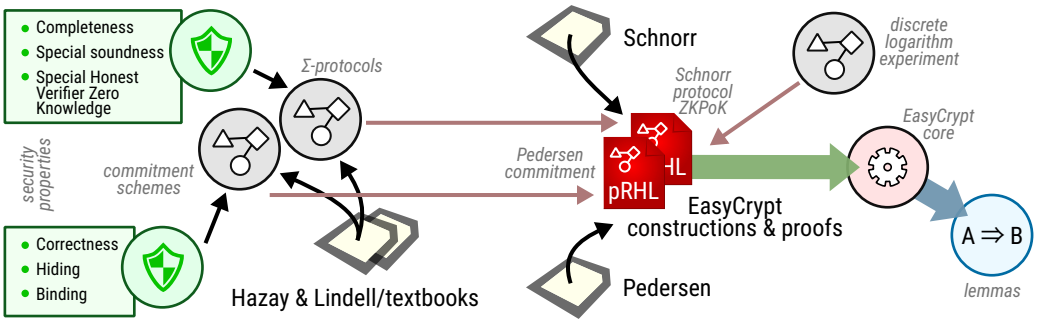
In this chapter we mechanise formal definitions of commitment schemes and sigma protocols, along with the security properties typical for those schemes, from Hazay and Lindell [99] and other textbooks [149]; a diagram of our contribution is illustrated in Figure 4.1. Then we show a computational model of them and illustrate the corresponding implementation in EasyCrypt, along with the mechanised proof of the security properties. To show the effectiveness of our mechanisation, we formally modelled and verified the Pedersen commitment scheme [136] and the Schnorr's protocol of Zero Knowledge Proof of Knowledge (ZKPoK) [145] by mean of cryptographic experiments and reductions to the discrete logarithm assumption of hardness.

4.1 Commitment schemes

In a commitment scheme, one wants to commit a message while preserving its secrecy for the time being until it is eventually revealed.

Our contribution is the support for generic commitment schemes, and our real-world motivating example is the popular Pedersen protocol.

Fig. 4.1 Description of our contribution for commitment schemes and Σ-protocols.



Commitment schemes are cryptographic primitives and their security is described as theorems that capture the internal behaviour of the protocol itself [99]. For this reason, the symbolic model is not suitable, as it treats cryptographic primitives as black boxes; therefore, we carried out our formalisation in the computational model. Some frameworks are available to work in such a model. CryptoVerif [45] is based on concurrent probabilistic process calculus. Although it is highly automatic, it is limited to prove properties related to secrecy and authenticity. The tool gga$^{\infty}$ [11] specialises in reasoning in the generic group model and seems promising when attackers have access to random oracles, which does not apply to our setting. Certicrypt is a fully machine-checked language-based framework built on top of the Coq proof assistant [27]. However it is no longer maintained. EasyCrypt [26] follows the same approach as CertiCrypt and supports automated proofs as well as interactive proofs that allow for interleaving both program verification and formalisation of mathematical theories. This is desirable because they are intimately intertwined when formalising cryptographic proofs, and can leave the tedious parts of proofs to machines.

Recently, a machine-checked formalisation of Σ-protocols to prove statements about discrete logarithms has been developed in CertiCrypt [28]. A commonality between the work in [28] and our work is that the Schnorr protocol proved in [28] is also based on the discrete logarithm assumption.

4.1.1 Definitions and properties

Commitments are very useful in secure computation, for example, in verifiable secret sharing, zero-knowledge proofs, and e-voting [131, 87].

The commitment must be bound to the original message, which means that the committer cannot change the message bound to the commitment once the message has been committed. Informally, the *hiding* property preserves the secrecy of the original statement to adversaries, while the *binding* property binds the commitment to its original value in the sense that finding a new value with the same commitment is infeasible, so it is not easy to cheat on the commitment.

More formally, a two-party commitment scheme is a protocol between a committer C and a receiver R which runs in two phases. Let M be the space of messages to commit to. The first phase is called *commitment phase*, where the party C sends R

its commitment for a private message $m \in M$ and secretly holds an opening value. The second phase is called *verification phase*, where the party C sends R the original message m along with the opening value, so that R can verify that the message committed in the first phase was indeed m.

Definition 4.1.1 (Commitment scheme)**.** *We define a commitment scheme π as the triplet $(\mathcal{G}, \mathcal{C}, \mathcal{V})$ such that:*

- *$\mathcal{G}\ (1^n)$ outputs a public value h;*
- *$\mathcal{C}\ (h, m)$ takes as input the public value h and the message m and outputs (c, d), where c is the commitment to send in the first phase and d is the opening value to be send in the second phase; and*
- *$\mathcal{V}\ (h, m, c, d)$ takes as input the public value h, the message m, the commitment c and the opening value d, and outputs* true *if verification succeeds or* false *otherwise.*

Let $\pi = (\mathcal{G}, \mathcal{C}, \mathcal{V})$ be a commitment scheme, its security properties are (i) correctness, i.e. for every message the commitment generated is valid, (ii) computational or perfect hiding, where any attacker cannot learn information from the commitment c about the message m with any advantage (perfect), or with a negligible advantage (computational), and (iii) computational or perfect binding, where the message m is uniquely bound to c (perfect) or finding another message with the same commitment has negligible probability of success (computational).

While correctness is defined without adversary, the other two properties are experiments played against any polynomially-bound adversary. Adversaries are probabilistic polynomial-time algorithms with **abstract** procedures. The adversary of the hiding experiment is called the *unhider*, while the adversary of the binding experiment is called the *binder*. Formally, we define the adversary $\mathcal{U}$ shaping the unhider which is required to have the following two procedures:

- $\mathcal{U}.choose$, accepting the public value h as argument and returning two messages in $\mathbb{Z}_q$, and
- $\mathcal{U}.guess$, accepting the commitment value c and returning a boolean value.

And we define the adversary $\mathcal{B}$ shaping the binder which is required to have the following procedure:

- $\mathcal{B}.bind$, accepting the public value h as argument and returning a quintuplet (c, m, d, m', d').

Constraining the adversary to have the above specific abstract procedures is not a restriction, since inside they can still call any kind of polinomially-bound routine of their choice. We base our formal definitions for all these security properties on algorithmic experiments that capture the security properties, see Figure 4.2.

Since the hiding and the binding experiments are defined against an adversary, they can be illustrated as adversary against a challenger, see Figure 4.3 and Figure 4.4.

Finally, the formal definitions based on the constructions above.

Fig. 4.2 Constructions for the security properties desired in commitment schemes.

$\mathsf{Corr}_{\pi,m}\,(1^n)$

1 $h \leftarrow \mathcal{G}\,(1^n)$;
2 $(c, d) \leftarrow \mathcal{C}\,(h, m)$;
3 $b \leftarrow \mathcal{V}\,(h, m, c, d)$;
4 **return** b

$\mathsf{HExp}_{\mathcal{U},\pi}\,(1^n)$

1 $h \leftarrow \mathcal{G}\,(1^n)$;
2 $(m_0, m_1) \leftarrow \mathcal{U}.choose\,(h)$;
3 $b \xleftarrow{\$} \{0, 1\}$;
4 $(c, \_) \leftarrow \mathcal{C}\,(h, m_b)$;
5 $b' \leftarrow \mathcal{U}.guess\,(c)$;
6 **return** $b = b'$

$\mathsf{BExp}_{\mathcal{B},\pi}\,(1^n)$

1 $h \leftarrow \mathcal{G}\,(1^n)$;
2 $(c, m, d, m', d') \leftarrow \mathcal{B}.bind\,(h)$;
3 $v \leftarrow \mathcal{V}\,(x, m, c, d)$;
4 $v' \leftarrow \mathcal{V}\,(x, m', c, d')$;
5 **return** $v \wedge v' \wedge m \neq m'$

Fig. 4.3 Alternative view of the hiding experiment, where the unhider $\mathcal{U}$ tries to win against its challenger.

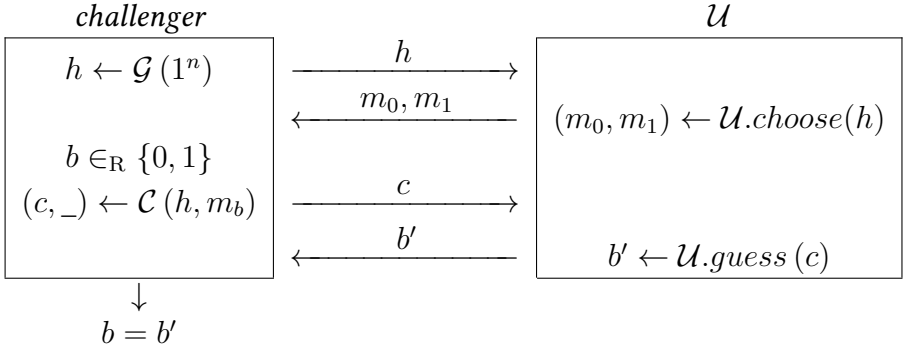
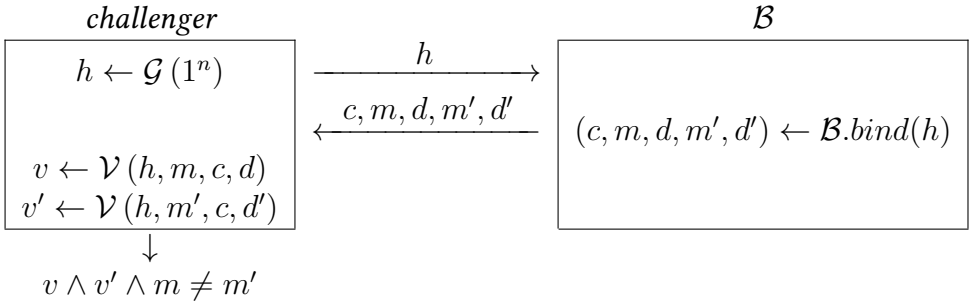


Fig. 4.4 Alternative view of the hiding experiment, where the binder $\mathcal{B}$ tries to win against its challenger.



Definition 4.1.2 (Correctness). *For all messages m in the message space M, we say that the scheme π is* correct *or* computationally correct *if the following are respectively true:*

$$\mathsf{Pr}\left[\mathrm{Corr}_{\pi,\mathrm{m}}\left(1^n\right)=1\right]=1 \tag{4.1}$$

$$\mathsf{Pr}\left[\mathrm{Corr}_{\pi,\mathrm{m}}\left(1^n\right)=1\right]\geq 1-\mu\left(n\right) \tag{4.2}$$

where μ is a negligible function on the security parameter n.

Definition 4.1.3 (Hiding). *For all probabilistic polynomial-time algorithms $\mathcal{U}$, we say that the scheme π satisfies the security property of* hiding *or* computational hiding *if the following are respectively true:*

$$\mathsf{Pr}\left[\mathsf{HExp}_{\mathcal{U},\pi}\left(1^n\right)=1\right]=\frac{1}{2} \tag{4.3}$$

$$\mathsf{Pr}\left[\mathsf{HExp}_{\mathcal{U},\pi}\left(1^n\right)=1\right]\leq\frac{1}{2}+\mu\left(n\right) \tag{4.4}$$

where μ is a negligible function on the security parameter n.

Definition 4.1.4 (Binding). *Let $\pi=(\mathcal{G},\mathcal{C},\mathcal{V})$ be a commitment protocol. Then we can define the binding properties for each polynomial time adversary $\mathcal{B}$.*

$$\begin{aligned}\text{(perfect binding)}\quad & \exists\,\mu.\ \mathsf{Pr}\left[\mathsf{BExp}_{\mathcal{B},\pi}\left(1^n\right)=1\right]=0\\ \text{(computational binding)}\quad & \exists\,\mu.\ \mathsf{Pr}\left[\mathsf{BExp}_{\mathcal{B},\pi}\left(1^n\right)=1\right]\leq\mu\left(n\right)\end{aligned}$$

where μ is a negligible function on the security parameter n.

4.1.2 Automatic verification of the Pedersen commitment scheme

Despite being a basic primitive in secure computation, to formalise it and have a computer generated proof is far from trivial. In the security proof generated by humans, many small gaps are left by the prover as they are easy to prove. However, for a machine the gaps can be huge and extra efforts need to be spent to let the machine complete the proof. In particular, to prove the perfect hiding property, we created a sequence of games that vary slightly to allow the machine to carry out the proof. This additional construction is absent from proofs in the original paper and is either absent or omitted in textbooks. In addition, to prove computational binding, we constructed a discrete logarithm game to allow for reduction.

Modelling the scheme and the properties

The abstract commitment scheme is modelled by the following few lines, prototyping the algorithms introduced in Section 4.1.1:

```
module type CScheme = { (* Abstract commitment scheme *)
  proc gen() : value
  proc commit(h: value, m: message) : commitment * openingkey
  proc verify(h: value, m: message, c: commitment, d: openingkey) : bool
}.
```

The Pedersen commitment protocol, shown in Figure 4.5, runs between the committer C, holding a secret message $m \in \mathbb{Z}_q$ to commit to, and the receiver R. Both agree on the group $(\mathbb{G}, q, g)$, where q is the order of $\mathbb{G}$ and g is its generator.

Commitment phase

- R samples a value $h \in_{\mathrm{R}} \mathbb{G}$ and sends it to C.
- C samples an opening value $d \in_{\mathrm{R}} \mathbb{Z}_q$, computes the commitment c as $g^d h^m$, and sends c to R.

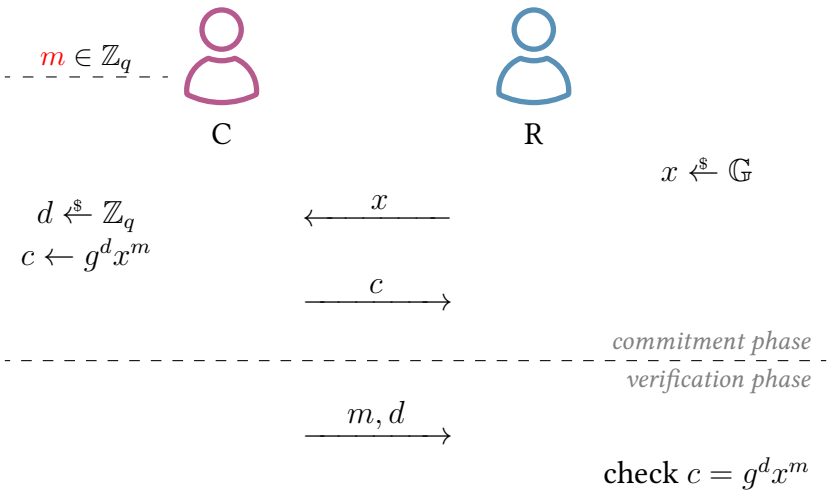
Verification phase

- C sends the pair (m, d) to R.
- R checks whether $g^d h^m$ matches to the previously received commitment c, and either *accepts* if they match or *reject* if the do not.

Its formal definition is given by the following algorithms:

$\mathcal{G}\ (1^n)$:	$\mathcal{C}\ (x, m)$:	$\mathcal{V}\ (x, m, c, d)$:
$x \leftarrow \mathbb{G}$;	$d \xleftarrow{\$} \{0,1\}_q$;	$c' \leftarrow g^d x^m$;
return x	$c \leftarrow g^d x^m$;	**return** $c = c'$
	return (c, d)	

Fig. 4.5 Pedersen commitment protocol.



We modelled the protocol in EasyCrypt as the following three procedures inside the module `module Ped : CScheme`:

```
proc gen() : value = {
  var x, h;
  x =$ FDistr.dt; (* This randomly samples an element in the field Z_q *)
  h = g^x; (* g is globally defined from the cyclic group theory *)
  return h;
}
proc commit(
    h: value, m: message
  ) : commitment * openingkey = {
  var c, d;
  d =$ FDistr.dt;
  c = (g^d) * (h^m);
  return (c, d);
}
```

```
proc verify(
    h: value, m: message,
    c: commitment, d: openingkey
  ) : bool = {
  var c';
  c' = (g^d) * (h^m);
  return (c = c');
}
```

The Pedersen commitment scheme security properties we prove are correctness, perfect hiding, and computational binding. Their model in EasyCrypt is shown in Figure 4.6. Those properties rely on the existence of a group $(\mathbb{G}, q, g)$ in which the discrete logarithm assumption holds (with regards to a secure parameter n):

$$\forall \mathcal{A}. \exists \mu. \quad \mathsf{Pr}\left[\mathsf{DLog}_{\mathcal{A}}\left(1^n\right) = 1\right] \leq \mu\left(n\right).$$

Fig. 4.6 Commitment scheme properties. Correctness (left), hiding experiment (middle) and binding experiment (right) modelled in EasyCrypt.

```
module Corr(S:CScheme) = {
 proc main(m: message)
    : bool = {
  var h, c, d, b;
  h = S.gen();
  (c, d) = S.commit(h, m);
  b = S.verify(h, m, c, d);
  return b;
 }                          }.
```

```
module HExp(
  S:CScheme, U:Unhider) = {
 proc main() : bool = {
  var b, b', m0, m1, h, c, d;
  h = S.gen();
  (m0, m1) = U.choose(h);
  b =$ {0,1};
  (c, d)=S.commit(h, b?m1:m0);
  b' = U.guess(c);
  return (b = b');
 }                          }.
```

```
module BExp(
  S:CScheme, B:Binder) = {
 proc main() : bool = {
  var h, c, m, m',
      d, d', v, v';
  h = S.gen();
  (c, m, d, m', d')=B.bind(h);
  v  = S.verify(h, m , c, d );
  v' = S.verify(h, m', c, d');
  return v /\ v' /\ (m <> m');
 }                          }.
```

The construction of the hiding experiment and of the binding experiment, introduced in Section 4.1.1 and modelled in Figure 4.6, can be instantiated with the Pedersen commitment scheme and result as in Figure 4.8.

Following such constructions, the desired security for this protocol is defined by the following two formulas, for all $\mathcal{U}$ and for all $\mathcal{B}$:

$$\begin{aligned} \text{(perfect hiding)} \quad & \mathsf{Pr}\left[\mathsf{HExp}_{\mathcal{U},\mathrm{Ped}}\left(1^n\right) = 1\right] = \frac{1}{2} \\ \text{(computational binding)} \quad & \exists \mu. \quad \mathsf{Pr}\left[\mathsf{BExp}_{\mathcal{B},\mathrm{Ped}}\left(1^n\right) = 1\right] \leq \mu\left(n\right) \end{aligned}$$

where μ is a negligible function on the security parameter n.

Correctness

Correctness in EasyCrypt is formalised with a HL judgement:

```
hoare[ Corr(Ped).main : T ⇒ res].
```

Its proof is straightforward. The first step is to unfolding the definition of `Corr(Ped).main`, which is the correctness algorithm described in Figure 4.6 instantiated with `Ped`. Then we have $c = g^d h^m$ and $c' = g^d h^m$ which are always equal.

Perfect hiding

In the Pedersen protocol we prove the perfect hiding:

$$\forall \mathcal{U}. \quad \mathsf{Pr}\left[\mathsf{HExp}_{\mathcal{U},\mathrm{Ped}}\left(\mathbb{G}, q, g\right) = 1\right] = \frac{1}{2} \tag{4.5}$$

In EasyCrypt, we modelled it with the following lemma:

```
lemma perfect_hiding: forall (U <: Unhider) &m,
  islossless U.choose ⇒ islossless U.guess ⇒
  Pr[HExp(Ped, U).main() @ &m : res] = 1%r / 2%r.
```

Where `U <: Unhider` is the adversary $\mathcal{U}$ with abstract procedures `choose` and `guess`, of which we needed to assume they terminate `islossless U.choose` and `islossless U.guess`.

Perfect hiding can be proved by comparing the hiding experiment to an intermediate experiment in which the commitment is replaced by g^d which contains no information about m_b. The experiment is described in Figure 4.7.

Fig. 4.7 The intermediate hiding experiment is almost equal to the hiding experiment, but the commitment is replaced by a random group element.

1 $\mathsf{HInterm}_{\mathcal{U},\mathsf{Ped}}(\mathbb{G}, q, g)$
2 $\quad h \xleftarrow{\$} \mathbb{G}$;
3 $\quad b \xleftarrow{\$} \{0,1\}$;
4 $\quad d \xleftarrow{\$} \mathbb{Z}_q$;
5 $\quad (m_0, m_1) \leftarrow \mathcal{U}.choose\,(h)$;
6 $\quad c \leftarrow g^d$; // msg independent
7 $\quad b' \leftarrow \mathcal{U}.guess\,(c)$;
8 $\quad$ **return** $b = b'$;

```
module HInterm(U:Unhider) = {
  proc main() : bool = {
    var b, b', x, c, d, m0, m1;
    x =$ FDistr.dt;
    b =$ {0,1};
    d =$ FDistr.dt;
    (m0, m1) = U.choose(g^x);
    c = g^d; (* message independent *)
    b' = U.guess(c);
    return (b = b');
    }
  }.
```

We prove it by first showing that for all adversaries, the probability of winning the hiding experiment is exactly the same as winning the intermediate experiment.

$$\forall \mathcal{U}. \quad \mathsf{Pr}\left[\mathsf{HExp}_{\mathcal{U},\mathrm{Ped}}(\mathbb{G}, q, g) = 1\right] = \mathsf{Pr}\left[\mathrm{HInterm}_{\mathcal{U},\mathrm{Ped}}(\mathbb{G}, q, g) = 1\right]$$

In code,

```
lemma phi_hinterm (U<:Unhider) &m:
  Pr[HExp(Ped,U).main() @ &m : res] = Pr[HInterm(U).main() @ &m : res].
```

To prove that, we unfold the two experiments in a pRHL judgement. The first experiment is automatically instantiated by EasyCrypt as the algorithm HExp in Figure 4.8.

The proof is done by comparing the execution of the two experiments and is based on the fact that the distribution of $h^{m_b} g^d$ is taken over g^d.

Then, we prove that for all adversaries, the probability of winning the intermediate experiment is exactly a half.

$$\forall \mathcal{U}. \quad \mathsf{Pr}\left[\mathrm{HInterm}_{\mathcal{U},\mathrm{Ped}}(\mathbb{G}, q, g) = 1\right] = \frac{1}{2}$$

In EasyCrypt, we have:

```
lemma hinterm_half (U<:Unhider) &m:
```

Fig. 4.8 Hiding experiment (left) and binding experiment (right) algorithms, instantiated with the Pedersen commitment scheme.

$\mathsf{HExp}_{\mathcal{U},\mathsf{Ped}}\,(\mathbb{G}, q, g)$

1. $h \in_{\mathrm{R}} \mathbb{G}$;
2. $b \in_{\mathrm{R}} \{0, 1\}$;
3. $d \in_{\mathrm{R}} \mathbb{Z}_q$;
4. $(m_0, m_1) \leftarrow \mathcal{U}.choose\,(h)$;
5. $c \leftarrow g^d h^{m_b}$;
6. $b' \leftarrow \mathcal{U}.guess\,(c)$;
7. **return** $b = b'$

$\mathsf{BExp}_{\mathcal{B},\mathsf{Ped}}\,(\mathbb{G}, q, g)$

1. $h \in_{\mathrm{R}} \mathbb{G}$;
2. $(c, m, d, m', d') \leftarrow \mathcal{B}.bind\,(h)$;
3. $v \leftarrow c = g^d h^m$
4. $v' \leftarrow c = g^{d'} h^{m'}$
5. **return** $v \wedge v' \wedge m \neq m'$

```
islossless U.choose ⇒ islossless U.guess ⇒
Pr[HInterm(U).main() @ &m : res] = 1%r/2%r.
```

Combining the two lemmas, by transitivity, we prove perfect hiding for Pedersen commitment protocol as in equation (4.5).

Computational binding

For the Pedersen protocol, we prove the computational binding property.

$$\forall\,\mathcal{B}.\ \exists\,\mu. \quad \mathsf{Pr}\left[\mathsf{BExp}_{\mathcal{B},\mathrm{Ped}}\,(\mathbb{G}, q, g) = 1\right] \leq \mu \tag{4.6}$$

where μ is a negligible function. The proof is done by a reduction to the discrete logarithm assumption. In cryptography, proof by reduction usually means to show how to transform an efficient adversary that is able to *break* the construction into an algorithm that efficiently solves a problem that is assumed to be hard. In this proof, the problem assumed to be hard is the discrete logarithm problem [109, p. 320]. We show that if an adversary can break the binding property, then it can output (m, d) and (m', d') such that $g^d h^m = g^{d'} h^{m'}$. If this is true then the discrete logarithm of $h = g^x$ can be computed by

$$x = \frac{d - d'}{m' - m}.$$

We capture the reduction by two modules in EasyCrypt, whose algorithms are illustrated in Figure 4.9. A small technical subtlety is that since the adversary is abstractly defined, it can return $m = m'$ with some probability. This can cause division by zero. Therefore, we check the output from the adversary to avoid it. Formally, the adversary assumed to break the binding experiment is $\mathcal{B}$ and we construct an adversary $\mathcal{A}$ to break the discrete logarithm experiment with equal probability of success:

$$\forall\,\mathcal{B}. \quad \mathsf{Pr}\left[\mathsf{BExp}_{\mathcal{B},\mathrm{Ped}}\,(\mathbb{G}, q, g) = 1\right] = \mathsf{Pr}\left[\mathsf{DLog}_{\mathcal{A}(\mathcal{B})}\,(\mathbb{G}, q, g) = 1\right]$$

The above is captured in EasyCrypt by the lemma:

```
lemma computational_binding: forall (B <: Binder) &m,
  Pr[BExp(Ped, B).main() @ &m : res] =
  Pr[DLog(DLogAttacker(B)).main() @ &m : res].
```

To prove the lemma, we unfolded the experiments as much as possible, i.e. up to abstractions, in a pRHL judgement which created an equivalence of the two experiments in the sense illustrated in Section 2.5. The binding experiment is automatically unfolded to the experiment BExp in Figure 4.8.

The automatic tactics could not automatically prove the lemma, as the expression $(d - d') / (m' - m)$ used by the attacker $\mathcal{A}$ (modelled as `DLogAttacker`) in the DLog experiment was too complex to be automatically used by the prover into the binding experiments and needed to be manually guided.

Assuming that the discrete logarithm is hard, then the probability of the experiment $\mathsf{BExp}_{\mathcal{B},\mathrm{Ped}}(\mathbb{G}, q, g)$ returning 1 must be negligible. Finally,

$$\forall \mathcal{B}.\ \exists \mu.\quad \Pr\left[\mathsf{BExp}_{\mathcal{B},\mathrm{Ped}}(\mathbb{G}, q, g) = 1\right] \leq \mu$$

which is the definition of computational binding we gave in equation (4.6).

Fig. 4.9 The discrete logarithm experiment (left) and an adversary reducing the binding experiment with the Pedersen protocol to the discrete logarithm experiment (right).

$\mathsf{DLog}_{\mathcal{A}}(\mathbb{G}, q, g)$

1. $x \in_{\mathrm{R}} \mathbb{Z}_q$;
2. $x' \leftarrow \mathcal{A}.\mathsf{guess}(g^x)$;
3. **if** $x' = \bot$ **then**
4. $\quad b \leftarrow \mathrm{false}$;
5. **else**
6. $\quad b \leftarrow (x' = x)$;
7. **return** b

$\mathcal{A}(\mathcal{B})$ .guess (h)

1. $(c, m, d, m', d') \leftarrow \mathcal{B}.bind(h)$;
2. **if** $c = g^d h^m = g^{d'} h^{m'} \wedge m \neq m'$ **then**
3. $\quad x \leftarrow \dfrac{d - d'}{m' - m}$;
4. **else**
5. $\quad x \leftarrow \bot$;
6. **return** x

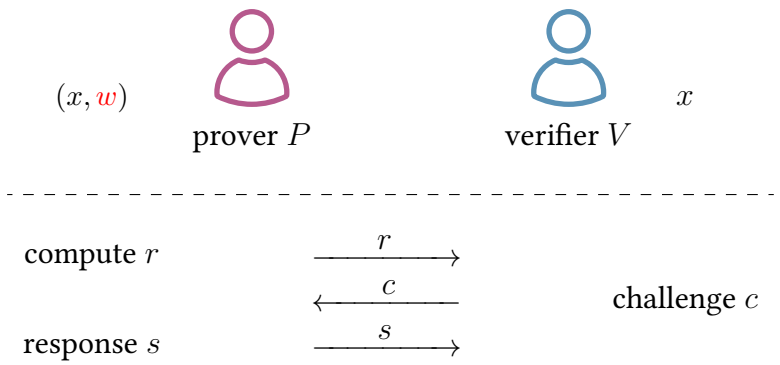
```
module DLog(A:Adversary) = {
  proc main () : bool = {
    var x, x', b;

    x =$ FDistr.dt;
    x' = A.guess(g^x);
    if (x' = None)
      b = false;
    else
      b = (x'= Some x);

    return b;
  }
}.
```

```
module DLogAttacker(B:Binder) : Adversary = {
  proc guess(h: group) : F.t option = {

    var x, c, m, m', d, d';
    (c, m, d, m', d') = B.bind(h);
    if ((c = g^d * h^m) /\
        (c = g^d' * h^m') /\ (m <> m'))
      x = Some((d - d') * inv (m' - m));
    else
      x = None;

    return x;
  }
}.
```

4.2 Σ protocols

Sigma protocols is a concept generalising those protocols that involve three sent-received-sent messages exchanged by two parties [68] whose scheme is illustrated in Figure 4.10.

Fig. 4.10 Sigma protocols.



Informally, they involve two parties P (called the prover) and V (called the verifier) who share a problem x whose solution w is known only to P, or, more generally, x and w are belong to some relation R. The first step is done by P who commits w to the value r and send it to V. In the second step, V challenges P sending her the challenge value c. In the last step, P sends its response s to the challenge back to V. Now, V verifies the response against x, r, and c. The goal of P is to convince V that she knows w, without actually disclosing it.

Sigma protocols present clear relation with commitment schemes, see Section 4.1, and an exhaustive comparison can be found in [99], i.e. how to build efficient commitment schemes from sigma protocols. They can enjoy previous formal verification [28] in a language called CertiCrypt (unmaintained an obsolete).c The formal proof done in CertiCrypt required about a few thousand lines of code, while in EasyCrypt only about 300 (we are considering only the abstraction supporting Sigma protocols and the Schnorr protocol as its instantiation). Clearly, the proof technique share similarities, yet they are different enough to be *freshly* re-done in EasyCrypt, since the two theorem provers are not compatible. We completed those proofs of security and pushed the EasyCrypt code online in the summer of 2017, but it has been merged to the official branch only recently in 2019[1].

4.2.1 Definitions and properties

Sigma protocols is a concept generalising those protocols that involve three sent-received-sent messages exchanged by two parties [68], of which scheme is illustrated in Figure 4.10. We borrow our definitions from Hazay and Lindell [99]. The aim of

[1] https://github.com/EasyCrypt/easycrypt/blob/1.0/theories/crypto/SigmaProtocol.ec

sigma protocols vary from proof of knowledge to proof of membership, and they are defined with respect to a binary relation R.

In this relation, given $(x, w) \in R$, w is called the *witness* to the statement x. The only property required to the relation $R \subset \{0,1\}^\star \times \{0,1\}^\star$ is that for all $(x, w, \in) R$, then the length of w is polynomial on the length of x. This restriction suggests that w can be thought as a (polynomial) solution to a computational problem x.

Informally, sigma protocols involve two parties P (called the prover) and V (called the verifier) who share a problem x whose solution w is known only to P, or, more generally, x and w are belong to some relation R.

The first step is done by P who commits w to the value r and send it to V. In the second step, V challenges P sending her the challenge c. In the last step, P sends its response s to the challenge back to V. Now, V verifies the response against x, r, and c. The goal of P is to convince V that she knows w, without actually disclosing it.

Definition 4.2.1 (Sigma protocol)**.** *Given a binary relation R, we define a sigma protocol Σ as the quintuplet $(\mathcal{G}, \mathcal{C}, \mathcal{T}, \mathcal{R}, \mathcal{V})$, run by a prover P and a verifier V, such that:*

- $(x, w) \leftarrow \mathcal{G}(1^n)$ *outputs a pair statement-witness $(x, w) \in R$, where x is publicly known and w is private to the prover;*
- $(m, \sigma) \leftarrow \mathcal{C}(x, w)$ *takes as input a statement x and a witness w and outputs a message m and a secret state σ that is hold by the prover;*
- $c \leftarrow \mathcal{T}(x, m)$ *takes as input a statement x and a message m and outputs a challenge c;*
- $r \leftarrow \mathcal{R}((x, w), (m, \sigma), c)$ *takes as input a statement-witness pair (x, w), a message-state pair (m, σ) and a challenge c, and outputs a response r; and*
- $\mathcal{V}(x, m, c, r)$ *takes as input a statement x, a message m, a challenge c and a response r and verifies that the $(x, w) \in R$ (this must be done by the verifier without knowing w).*

Sigma protocols present clear relation with commitment schemes, and an exhaustive comparison can be found in [99], i.e. how to build efficient commitment schemes from sigma protocols. We refer to the the triplet of the message, challenge and response produced by a run of a sigma protocol as a *transcript*.

Definition 4.2.2 (Transcript)**.** *A sigma protocol $\Sigma = (\mathcal{G}, \mathcal{C}, \mathcal{T}, \mathcal{R}, \mathcal{V})$ for the relation R, run by a prover P that knows $(x, w) \in R$ and a verifier V that knows x, produces a* transcript (m, c, r) *where m, c and r are the messages exchanged between R and V. In an honest execution, they are*

$$\begin{aligned} m, \_ &\leftarrow \mathcal{C}(x, w) \\ c &\leftarrow \mathcal{T}(x, m) \\ r &\leftarrow \mathcal{R}((x, w), (m, \sigma), c). \end{aligned}$$

The desired properties of sigma protocols are the following:

Completeness The verifier always accepts valid statement-witness pairs (valid pairs are those belonging to the relation R).

Special soundness It is efficient to find the witness w corresponding to an x, i.e. such that $(x, w) \in R$, given two transcripts of the same initial message r with different challenge c.

Special honest verifier zero knowledge The protocol can be simulated by only knowing the known initial value x (the problem) and the challenge, in such a way c that the transcript generated by the simulator has the same probability of the sigma protocol run by two hones parties on x.

Those properties are captured by the Definitions 4.2.3, 4.2.4, and 4.2.5 respectively.

We base our formal definitions for all these security properties on algorithmic experiments that capture the security properties, see Figure 4.11.

Fig. 4.11 Constructions for the security properties desired in Σ protocols.

$\mathsf{Completeness}_{\Sigma}\left(1^n\right)$
1. $(x, w) \leftarrow \mathcal{G}\left(1^n\right)$;
2. $(m, \sigma) \leftarrow \mathcal{C}\left(x, w\right)$;
3. $c \leftarrow \mathcal{T}\left(x, w\right)$;
4. $r \leftarrow \mathcal{R}\left(\left(x, w\right), \left(m, \sigma\right), c\right)$;
5. $b \leftarrow \mathcal{V}\left(x, m, c, r\right)$;
6. **return** b

$\mathsf{Soundness}_{\Sigma}^{\mathcal{A}}\left(x, t, t'\right)$
1. $w \leftarrow \mathcal{A}\left(t, t'\right)$;
2. $(m, c, r) \leftarrow t$;
3. $(m', c', r') \leftarrow t'$;
4. $v \leftarrow \mathcal{V}\left(x, m, c, r\right)$;
5. $v' \leftarrow \mathcal{V}\left(x, m', c', r'\right)$;
6. $h \leftarrow c \neq c' \wedge m = m'$;
7. **return** $h \wedge v \wedge v' \wedge (x, w) \in R$

$\mathsf{SpecialHVZK}_{\Sigma,\mathsf{Sim}}^{\mathcal{A}}\left(1^n\right)$
1. $(x, w) \leftarrow \mathcal{G}\left(1^n\right)$;
2. $c \xleftarrow{\$} \delta$;
3. $b \xleftarrow{\$} \{0, 1\}$;
4. **if** b **then**
5. $\quad m, \sigma \leftarrow \mathcal{C}\left(x, w\right)$;
6. $\quad r \leftarrow \mathcal{R}\left(\left(x, w\right), \left(m, \sigma\right), c\right)$;
7. **else**
8. $\quad m, c, r \leftarrow \mathsf{Sim}\left(x, c\right)$;
9. $b' \leftarrow \mathcal{A}\left(m, c, r\right)$;
10. **return** $b = b'$

Definition 4.2.3 (Completeness). *Given a sigma protocol Σ for the relation R, run by a* honest *prover P and a* honest *verifier V, then Σ respects the property of* completeness *if and only if*

$$\mathsf{Pr}\left[\mathsf{Completeness}_{\Sigma}\left(1^n\right) = 1\right] = 1.$$

Definition 4.2.4 (Special soundness). *A sigma protocol Σ for the relation R, run by a prover P and a verifier V over the statement x, respects the property of* special soundness *if and only if*

$$\mathsf{Pr}\left[\mathsf{SpecialSoundness}_{\Sigma}^{\mathcal{A}}\left(x, t, t'\right) = 1\right] = 1,$$

where $\mathcal{A}$ is a polynomial-time algorithm, and t and t' are two (accepting) transcripts of an execution of Σ over the same statement x.

Special soundness ensures that a Σ protocol is an *interactive* proof, i.e. its result cannot be re-used or the knowledge cannot be transferred.

Definition 4.2.5 (Special honest verifier zero knowledge)**.** *Given a sigma protocol $\Sigma = (\mathcal{G}, \mathcal{C}, \mathcal{T}, \mathcal{R}, \mathcal{V})$ for the relation R, run by a prover P and a verifier V over the statement x, then Σ respects the property of* special honest verifier zero knowledge *if and only if exists a polynomial-time simulator* Sim *such that*

$$\mathsf{Pr}\left[\mathsf{SpecialHVZK}^{\mathcal{A}}_{\Sigma,\mathsf{Sim}}\left(1^n\right) = 1\right] = \frac{1}{2} + \mu\left(n\right),$$

where $\mathcal{A}$ is a polynomial-time algorithm, t and t' are two (accepting) transcripts of an execution of Σ over the same statement x, and μ is a negligible function on the security parameter n.

Special honest verifier zero knowledge ensures that an adversary $\mathcal{A}$ external to the protocol cannot infer any knowledge about the witness, including the result, i.e. whether the prover knows the witness or not.

4.2.2 Automatic verification of the Schnorr protocol

The Schnorr protocol for zero-knowledge proof of knowledge relies on the hardness of discrete logarithm since it instantiates the statement x and the witness w as $x = g^w$. The statement x can be interpreted as the problem of computing discreet logarithm, whose solution is w. Since the discreet logarithm is hard to compute, with the Schnorr protocol a prover can demonstrate the knowledge of w, given x: hardly, any adversary would be able to prove the same.

Modelling the scheme and the properties

The abstract sigma protocol scheme is modelled by the following few lines, prototyping the algorithms introduced in Section 4.2.1:

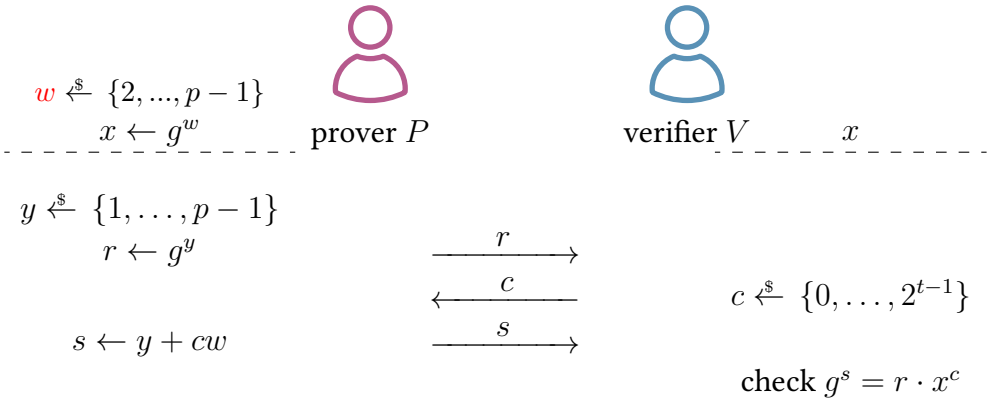
```
module type SigmaScheme = {
  proc gen() : statement * witness
  proc commit(x: statement, w: witness) : message * prover_state
  proc test(x: statement, m: message) : challenge
  proc respond(sw: statement * witness, ms: message * prover_state, e: challenge) : response
  proc verify(x: statement, m: message, e: challenge, z: response) : bool
}.
```

The Schnorr protocol, illustrated in Figure 4.12, runs between the prover V, holding a secret witness $w \in \{2, ..., p-1\}$, and the prover B, where both know $x = g^w$. Both agree on the group $(\mathbb{G}, q, g)$, where q is the order of $\mathbb{G}$ and g is its generator.

The Schnorr protocol runs the following way

- P samples a value y uniformly at random from $\mathbb{Z}_q^\star$ and sends it to V.

Fig. 4.12 Schnorr's zero-knowledge proof of knowledge. The values t, p, and g are accepted public knowledge, where t is the length of the challenge, p is a prime number and g is a generator of $\mathbb{Z}_q^\star$.



- V generates a challenge c by sampling at ranomd from $\{0, \ldots, 2^{t-1}\}$, and sends it to P.
- Finally, P responds with s to V, where $s = y + cw$, and the verifier V can check whether $g^s = r \cdot x^c$.

Its formal definition is given by the following algorithms:

$\mathcal{G}\ (1^n)$:
  $w \xleftarrow{\$} \{2, ..., p-1\}$;
  $x \leftarrow g^w$;
  return (x, w)

$\mathcal{C}\ (x, w)$:
  $y \xleftarrow{\$} \{1, \ldots, p-1\}$;
  $r \leftarrow g^y$;
  return (r, y)

$\mathcal{T}\ (x, r)$:
  $c \xleftarrow{\$} \{0, \ldots, 2^{t-1}\}$;
  return c

$\mathcal{R}\ ((x, w), (r, y), c)$:
  $s \leftarrow y + cw$;
  return s

$\mathcal{V}\ (x, r, c, s)$:
  return $g^s = r \cdot x^c$

We modelled the protocol in EasyCrypt as the following file procedures inside the module `module SchnorrPK : SigmaScheme`:

```
proc gen() : statement * witness = {
  var x, w;
  w <$ FDistr.dt \ F.zero;
  x = g^w;
  return (x, w);
}

proc commit(x: statement, w: witness) : message * secret = {
  var r, y;
  y <$ FDistr.dt;
  r = g^y;
  return (r, y);
}

proc test(x: statement, r: message) : challenge = {
```

```
  var c;
  c <$ FDistr.dt;
  return c;
}

proc respond(xw: statement * witness, ry: message * secret, c: challenge) : response = {
  var s, w, y;
  w = snd xw;
  y = snd ry;
  s = y + c*w;
  return s;
}

proc verify(x: statement, r: message, c: challenge, s: response) : bool = {
  var v, v';
  v  = r*(x^c);
  v' = g^s;
  return (v = v');
}
```

We proved the completeness, special soundness and special honest verifier zero knowledge security properties for the Schnorr protocol as follows. Their model in EasyCrypt is shown in Figure 4.13. Those properties rely on the existence of a group $(\mathbb{G}, q, g)$ in which the discrete logarithm assumption holds (with regards to a secure parameter n):

$$\forall \mathcal{A}. \exists \mu. \quad \mathsf{Pr}\left[\mathsf{DLog}_{\mathcal{A}}\left(1^n\right) = 1\right] \leq \mu\left(n\right).$$

The construction of the completeness experiments, the special soundness experiment and of the special honest verifier zero knowledge experiment, introduced in Section 4.2.1 and modelled in Figure 4.13, can be instantiated with the Schnorr protocol illustrated in Figure 4.14. Through cryptographic experiments, we rigorously re-formalise the proofs of Schnorr.

Following such constructions, the desired security for this protocol is defined by the following three formulas, for all algorithms $\mathcal{A}$:

$$\begin{aligned}
\text{(completeness)} \quad & \mathsf{Pr}\left[\mathsf{Completeness}_{\mathsf{Schnorr}}\left(x, w\right) = 1\right] = 1, \\
\text{(special soundness)} \quad & \mathsf{Pr}\left[\mathsf{SpecialSoundness}_{\mathcal{A},\mathsf{Schnorr}}\left(x, t, t'\right) = w \wedge \left(x, w\right) \in R\right] = 1, \\
\text{(special HVZK)} \quad & \exists \mu. \quad \mathsf{Pr}\left[\mathsf{SpecialHVZK}^{\mathcal{A}}_{\mathsf{Schnorr},\mathsf{Sim}}\left(1^n\right) = 1\right] = \frac{1}{2} + \mu\left(n\right),
\end{aligned}$$

where μ is a negligible function on the security parameter n. We remark that the algorithm $\mathcal{A}$ for the special soundness is not an adversarial entity, but it is an algorithm that efficiently recovers the witness given two accepting transcripts with the same challenge. This ensures that the protocol is an *interactive* proof, i.e. its result cannot be re-used or the knowledge cannot be transferred. Very differently, the special honest verifier zero knowledge is a simulation based proof, as introduced in Section 2.2.3, where the adversarial entity $\mathcal{A}$ tries to distinguish between the real construction and an ideal, simulated construction.

Completeness

Completeness in EasyCrypt is formalised with a HL judgement:

Fig. 4.13 Security properties of Sigma protocols. Experiments modelled in EasyCrypt that capture completeness, special soundness and special honest verifier zero knowledge.

```
module Completeness (S: SigmaScheme) = {
  proc main(x: statement, w: witness) : bool = {
    var e, m, s, z, b;
    (m, s) <@ S.commit(x, w);
    e      <@ S.test(x, m);
    z      <@ S.respond((x, w), (m, s), e);
    b      <@ S.verify(x, m, e, z);
    return b; } }.

module SpecialSoundnessExperiment (S: SigmaScheme, A: SigmaAlgorithms) = {
  proc main(x: statement, m: message,
      e: challenge, z: response,
      e': challenge, z': response) : witness option = {
    var s, sto, w, r, v, v';
    sto <@ A.soundness(x, m, e, z, e', z');
    if (sto <> None) {
      w  = oget sto;
      v  <@ S.verify(x, m, e , z  );
      v' <@ S.verify(x, m, e', z' );
      r  = R x w;
      if (e <> e' /\ r /\ v /\ v') {
        s = Some(w);
      } else {
        s = None;
      }
    } else {
      s = None;
    }
    return s; } }.

module SpecialHVZK (S: SigmaScheme, A: SigmaAlgorithms, D: SigmaTraceDistinguisher) = {
  proc gameIdeal() : bool = {
    var b, e, i, m, s, t, to, x, w;
    (x, w) <@ S.gen();
    (m, s) <@ S.commit(x, w);
    e      <$ de;
    to     <@ A.simulate(x, e);
    i      = 0;
    while (to = None) {
      to <@ A.simulate(x, e);
      i  = i + 1;
    }
    return oget to;
  }

  proc gameReal() : bool = {
    var b, t, m, e, z, x;
    (x, m, e, z) <@ Run(S).main();
    return (m, e, z);
  }

  proc main() : bool = {
    var b, b';
    b <$ {0,1};
    if (b) {
      t  <@ gameIdeal();
      b' <@ D.distinguish(x, t);
    } else {
      t  <@ gameReal();
      b' <@ D.distinguish(x, t);
    }
    return (b = b');
  }
}.
```

Fig. 4.14 Completeness experiment, special soundness experiment and special honest verifier zero knowledge experiment, instantiated with the Schnorr protocol.

$\mathsf{Completeness}_{\mathsf{Schnorr}}\ (x, w)$

1. $y \xleftarrow{\$} \{1, \ldots, p-1\}$;
2. $r \leftarrow g^y$;
3. $c \xleftarrow{\$} \{0, \ldots, 2^{t-1}\}$;
4. $s \leftarrow y + cw$;
5. $b \leftarrow g^s = r \cdot x^c$;
6. **return** b

$\mathsf{SpecialSoundness}_{\mathcal{A},\mathsf{Schnorr}}\ (x, t, t')$

1. $w \leftarrow \mathcal{A}\,(t, t')$;
2. $(r, c, s) \leftarrow t$;
3. $(r', c', s') \leftarrow t'$;
4. $v \leftarrow s = r \cdot x^c$;
5. $v' \leftarrow s' = r' \cdot x^{c'}$;
6. $h \leftarrow c \neq c' \wedge m = m'$;
7. **if** $h \wedge v \wedge v'$ **then**
8. **return** w
9. **else**
10. **return** $\perp$

$\mathsf{SpecialHVZK}^{\mathcal{A}}_{\mathsf{Schnorr},\mathsf{Sim}}\ (1^n)$

1. $w \xleftarrow{\$} \{2, ..., p-1\}$;
2. $x \leftarrow g^w$;
3. $c \xleftarrow{\$} \{1, \ldots, 2^{t-1}\}$;
4. $b \xleftarrow{\$} \{0, 1\}$;
5. **if** b **then**
6. $y \xleftarrow{\$} \{1, \ldots, p-1\}$;
7. $r \leftarrow g^y$;
8. $s \leftarrow y + cw$;
9. **else**
10. $r, c, s \leftarrow \mathsf{Sim}\,(x, c)$;
11. $b' \leftarrow \mathcal{A}\,(r, c, s)$;
12. **return** $b = b'$

```
lemma schnorr_completeness x w &m:
  R x w =>
  Pr[Completeness(SchnorrPK).main(x, w) @ &m : res] = 1%r.
```

Its proof is straightforward. The first step is to unfolding the definition of `Completeness(Schnorr).main`, which is the completeness algorithm described in Figure 4.14 instantiated with `Schnorr`. Then we have $s = y + cw$ as an assignment, then the verification checks if the equality $g^s = r \cdot x^c$ holds. Let us develop g^s substituting s with its assignment and show the algebra steps to prove the equality.

$$g^s = g^{y+cw} = g^y \cdot g^{cw} = r \cdot (g^w)^c = r \cdot x^c,$$

where $r = g^y$ and $x = g^w$ by previous assignments. So the verification is always accepting under the premise $(x, w) \in R$.

Special soundness

In the Schnorr protocol we prove the special soundness:

$$\Pr\left[\mathsf{SpecialSoundness}_{\mathcal{A},\mathsf{Schnorr}}\,(x, t, t') = w \wedge (x, w) \in R\right] = 1 \tag{4.7}$$

In EasyCrypt, we modelled it with the following lemma:

```
lemma schnorr_special_soundness (x: statement) r c c' s s' &m:
```

```
c <> c' =>
g^s  = r*(x^c ) =>
g^s' = r*(x^c') =>
Pr[SpecialSoundnessExperiment(SchnorrPK, SchnorrPKAlgorithms)
  .main(x, r, c, s, c', s') @ &m :
  (res <> None /\ R x (oget res))] = 1%r.
```

Where `ch <> ch'` is the premise stating that the challenges differ, `g^r = msg*(h^ch )` and `g^r' = msg*(h^ch')` are the premises ensuring that the transcripts are accepted. The two accepting transcripts are (r, c, s) and (r, c', s').

Showing that this lemma holds can be done with algebra steps, similarly to what we have done for the completeness. The core calculus of this proof may recall a small part of the proof of binding for the Pedersen commitment scheme, see Section 4.1.2 for a comparison.

The algorithm illustrated in Figure 4.15 is efficient and produces a witness w such that $(x, w) \in R$.

Fig. 4.15 The algorithm $\mathcal{A}$ to prove special soundness of the Schnorr protocol.

1 $\mathcal{A}\ (x, t, t')$:
2 $(r, c, s) \leftarrow t$;
3 $(r', c', s') \leftarrow t'$;
4 $v \leftarrow g^s = r \cdot x^c$;
5 $v' \leftarrow g^{s'} = r' \cdot x^{c'}$;
6 **if** $r = r' \wedge c \neq c' \wedge v \wedge v'$ **then**
7 $w \leftarrow \dfrac{s - s'}{c - c'}$;
8 **return** w
9 **else**
10 **return** $\perp$

```
module SchnorrPKAlgorithms {
[···]
proc soundness(
    x: statement, r: message,
    c: challenge, s: response,
    c': challenge, c': response
  ) : witness option = {
  var sto, w, v, v';
  v  = (g^s  = r*(x^c ));
  v' = (g^s' = r*(x^c'));
  if (c <> c' /\ v /\ v') {
    w = (s - s') / (c - c');
    sto = Some(w);
  } else {
    sto = None;
  }
  return sto; } }
```

To prove the validity of the algorithm, we first use the two premises of accepting transcripts.

$$g^s = r \cdot x^c,$$
$$g^{s'} = r \cdot x^{c'}.$$

In summary, the two transcripts $t = (r, c, s)$ and $t' = (r', c', s')$ passes to $\mathcal{A}$ have the same commitment $r = r'$, different challenge $c \neq c'$ and such that the transcripts are

accepting. So, if we divide the two accepting equations, we have:

$$\begin{aligned}\frac{g^s}{g^{s'}} &= \frac{r \cdot x^c}{r \cdot x^{c'}} \\ g^{s-s'} &= \frac{x^c}{x^{c'}} \\ g^{s-s'} &= x^{c-c'}.\end{aligned}$$

Since we need to find a w such that $(x, w) \in R$, and the relation in the Schnorr protocol is $x = g^w$, we make the substitution and write

$$\begin{aligned}g^{s-s'} &= (g^w)^{c-c'} \\ g^{s-s'} &= g^{w(c-c')}.\end{aligned}$$

Now we apply the discreet logarithm $\log$ to both hands of the equality and write

$$\begin{aligned}\log\left(g^{s-s'}\right) &= \log\left(g^{w(c-c')}\right) \\ s - s' &= w\left(c - c'\right) \\ w &= \frac{s - s'}{c - c'},\end{aligned}$$

where we applied the theorem for which $\forall x, \log\left(g^x\right) = x$, and the last step is always legitimate as the denominator is non-null, $c \neq c'$ hence $c - c' \neq 0$. So the assumptions that the two transcripts are accepting and $c \neq c'$ are enough for the existence of a polynomial algorithm $\mathcal{A}$ to *always* reconstruct a valid witness, thus to achieve the special soundness for the Schnorr protocol.

Special honest verifier zero knowledge

For the Schnorr protocol, we prove the special honest verifier zero knowledge property. Exists a simulator Sim such that, for any polynomial-time distinguisher $\mathcal{A}$, there exists a negligible function μ such that

$$\Pr\left[\mathsf{SpecialHVZK}^{\mathcal{A}}_{\mathsf{Schnorr},\mathsf{Sim}}\left(1^n\right) = 1\right] \leq \mu\left(n\right) \tag{4.8}$$

where n is the security parameter. In EasyCrypt we capture the security property with

```
lemma schnorr_shvzk (D<: SigmaTraceDistinguisher) &m:
  Pr[SpecialHVZK
      (SchnorrPK, SchnorrPKAlgorithms, D).main() @ &m : res] = 1%r/2%r.
```

The simulator Sim is illustrated in Figure 4.16 and its output samples the response s, then reconstructs the message r using the challenge c, inverting the operation that would have been done sampling r first and then computing the response s with the challenge c.

Fig. 4.16 The simulator Sim to prove special honest verifier zero knowledge of the Schnorr protocol.

1 Sim (x, c):
2 $\quad s \xleftarrow{\$} \mathbb{Z}_q$;
3 $\quad r \leftarrow \frac{g^s}{x^c}$;
4 $\quad$ **return** (r, c, s)

```
module SchnorrPKAlgorithms {
[···]
proc simulate(x: statement, c: challenge)
    : message * challenge * response = {
  var r, s;

  s  <$ FDistr.dt;
  r  = (g^s) * (x^(-c));

  return (r, c, s); } }
```

The proof is done by showing that the transcript $t' = (r', c', s')$ produced by the simulator and the transcript $t = (r, c, s)$ produced by a real execution of the Schnorr protocol are statistically equivalent, $t \overset{s}{\equiv} t'$.

To do that we first show that each parts of the transcripts are statistically equivalent, i.e. $r \overset{s}{\equiv} r'$, $c \overset{s}{\equiv} c'$ and $s \overset{s}{\equiv} s'$. Then, we show that their relationships are the same.

First we show the equivalence of the challenges. Following the experiment illustrated in Figure 4.14, we notice that $c = c'$, which trivially implies their equivalence $c \overset{s}{\equiv} c'$.

We remain to prove the equivalences of the messages and the responses. In the real execution, s (response) is produced after r (message), which depends on y (secret state of the prover). First, y is sampled from $\mathbb{Z}_q$, then $r \leftarrow g^y$ and finally $s \leftarrow y + cw$. Differently in the simulation, r' is produced after s'. First, s' is sampled from $\mathbb{Z}_q$, then $r' \leftarrow g^{s'} x^{-c}$.

For the equivalence of the responses, we notice that clearly $s' \overset{s}{\equiv} y$, as they follow the same distribution. Also, $y + cw$ has the same probability as y of being sampled from $\mathbb{Z}_q$, implying that $s \overset{s}{\equiv} y$. Transitivity of $\overset{s}{\equiv}$ finally implies that $s \overset{s}{\equiv} s'$.

The last equivalence to show is $r \overset{s}{\equiv} r'$. It is easy to see that

$$r \overset{s}{\equiv} r' \cdot x^c$$

Now, if we substitute r with g^y and x with g^w, we have

$$\begin{aligned} & g^y \overset{s}{\equiv} r' \cdot g^{cw} \\ \Rightarrow \quad & g^y g^{-cw} \overset{s}{\equiv} r' \\ \Rightarrow \quad & g^{y-cw} \overset{s}{\equiv} r' \\ \Rightarrow \quad & y - cw \overset{s}{\equiv} log\,(r')\,. \end{aligned}$$

where in the last equivalence we applied the theorem $\forall x, \log(g^x) = x$ after having applied the $\log$ to both hands of the equivalence. At this point we notice that y and $y - cw$ have the same probability of being sampled, so $y \overset{s}{\equiv} y - cw$, then

$$y \overset{s}{\equiv} log(r') \Rightarrow g^y \overset{s}{\equiv} r'.$$

Substituting back g^y with r, we have demonstrated that $r \overset{s}{\equiv} r'$.

At this point, we show that the relationship between the elements of the triplet are consistent. In particular, it is easy to see that

$$g^s = rg^{cw} \text{ and } g^{s'} = r'g^{cw}.$$

This completes the proof that the two transcripts are statistically equivalent, $t \overset{s}{\equiv} t'$. Therefore, the Schnorr protocol illustrated in Figure 4.12 respects the property of special honest verifier zero knowledge. This proof follows the original one from Schnorr, that we adapted to explicitly use cryptographic games to formalise the reductions.

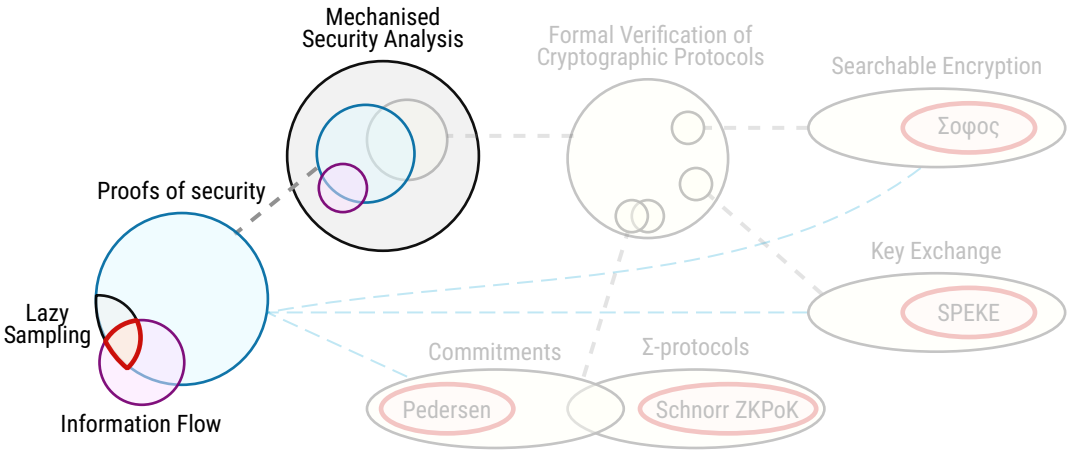
4.3 Conclusion

We extended the corpus of theories of EasyCrypt with commitment schemes and Σ protocols, by mechanising their security properties following the theory from Hazay and Lindell [99]. To show the effectiveness or the mechanisation, we propose an automated verification of the Pedersen commitment scheme and the Schnorr protocol of zero knowledge proof of knowledge. We adapted the proofs to be captured as cryptographic experiments; this also simplifies the formalisation of the reductions to cryptographic hardness assumptions. Even though we did not find significant gaps in the formal analysis on papers, our work strengthens their confidence and the theorems can be re-used for different cryptographic schemes that use commitment schemes or sigma protocols as cryptographic primitives. We analysed and successfully mechanised the proof of the security properties of the above mentioned protocols:

- perfect hiding and computational binding for the Pedersen commitment scheme, and
- completeness, special soundness and special honest verifier zero knowledge for the Schnorr protocol.

Chapter 5

Information flow in the pRHL



In this chapter, we address the problem of indistiguishability between two lazy constructions that typically raise in formal proofs that involve random oracles. We propose a proof strategy that may simplify proofs if related to the currently adopted approach by introducing information flow labels to variables. In detail, we introduce new syntax and tactics at the core of EasyCrypt and finally show a case study were we prove indistinguishability between two lazy constructions that use an internal map that emulate a random function.

5.1 Introduction

A significant amount of cryptographic proofs are based on the concept of indistinguishability between two different algorithms c_1 and c_2, denoted as $c_1 \sim c_2$, by an attacker who decides their initial inputs, inspects their final outputs, but is unaware of their internal construction. These proofs can be very complex and articulated. To break down their complexity, intermediate subsequent constructions are created and differ by very little one another, e.g. few lines of code. Once the indistinguishability between all the adjacent constructions is demonstrated, they are finally combined to prove the original statement. One case of indistinguishability between two subsequent constructions is when a probabilistic assignment moves from a procedure

to another. Figure 5.1 illustrates one such a case, where M_1 and M_2 are two very similar constructions that include the procedures $init$, f, and g. The implementation of both M_1 and M_2 perform *lazy sampling*, i.e. instead of making random choices upfront, they delay making random choices until they are actually needed [39]. The

Fig. 5.1 Two constructions M_1 and M_2; they differ only on the implementation of g.

$\underline{M_1}$:	$\underline{M_2}$:
$v \in \{0,1\} \cup \bot$	$v \in \{0,1\} \cup \bot$
proc $init()$	**proc** $init()$
1 $v \leftarrow \bot$;	1 $v \leftarrow \bot$;
proc $f()$	**proc** $f()$
2 **if** $v = \bot$ **then**	2 **if** $v = \bot$ **then**
3 $v \xleftarrow{\$} \{0,1\}$;	3 $v \xleftarrow{\$} \{0,1\}$;
4 **return** v	4 **return** v
proc $g()$	**proc** $g()$
5 **if** $v = \bot$ **then**	5 **return**
6 $v \xleftarrow{\$} \{0,1\}$;	
7 **return**	

procedure $init$ is called once to initialise the value v. The difference between the procedures f and g is that the latter does not return a value. M_1 and M_2 differ in the internal behaviour of the procedure g. In detail, the probabilistic assignment of v disappears from $M_2.g$ which can only be done when $M_2.f$ is called. Upon calling f the first time after having called g, the value v is freshly sampled in M_2 but already stored in M_1.

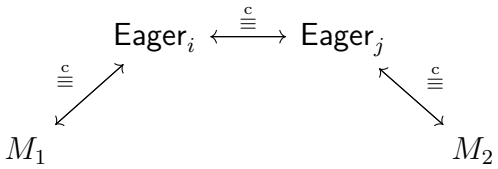
The adversary cannot call $init$ nor access v directly but can call f and g a polynomial number of times and inspect the output. This kind of situation in formal proof is common when constructing simulators where structures depending on private values are simulated by random values or random oracles.

We focus on the indistiguishability between M_1 and M_2, denoted as $M_1 \overset{c}{\equiv} M_2$. During a proof of indistiguishability between any two constructions, one must show that the (even small) differences will not affect the distribution of the output to lead the adversary to tell the constructions apart. M_1 and M_2 differ in their internal behaviour, so if the adversary calls g before f, then $M_1.f$ would simply return a stored value, while $M_2.f$ would first sample it from $\{0,1\}$. Informally, at some point in the proof, we would encounter a proof obligation with the following structure.

$$\begin{aligned} \Psi \Rightarrow & \\ & \textbf{return } M_1.v \quad \sim \quad M_2.v \xleftarrow{\$} \{0,1\}\,; \\ & \qquad\qquad\qquad\quad \textbf{return } M_2.v \\ \Rightarrow\ & M_1.v \simeq M_2.v \end{aligned} \tag{5.1}$$

where Ψ is some precondition at that point of the proof flow that depends on previous steps, and at the end we need to show that $M_1.v$ and $M_2.v$ are equally distributed. A more formal notation will be introduced later in the discussion. However generally, $M_1.v$ and $M_2.v$ are not equally distributed, e.g. if $M_1.g$ deterministically assigned 1 to v instead of sampling from $\{0, 1\}$, then M_1 and M_2 would *not* be indistiguishable.

To discharge the above proof obligation, one can reason about to the *history* of the value v, on whether it was used from its last sampling or not. In the literature, the problem in the proof obligation has been addressed in proofs of equivalence between a *lazy sampling* and *eager sampling* [39]. The lazy sampling technique delays the random sampling of a value until the point in the flow of a program execution where it is first used. Differently, the eager sampling technique chooses the same value at random before the execution. Therefore, to prove the indistiguishability between M_1 and M_2, one could start building their equivalent eager games, Eager_1 and Eager_2 then use the transitivity property to complete the proof.



We call this the *eager-lazy approach* and a mechanised example of this technique has been showed by Barthe et al. [27] and it is the state-of-the-art approach to indistinguishability proofs between lazy constructions [27, 154, 10].

The core idea is that if v has not been used in the game since it was last sampled, then it is perfectly fine to resample it. Despite of this simple intuition, the actual implementation that mechanises this concept is far from trivial. To appreciate the non-triviality of such approach, we discuss the core lemma that allows one to substitute an already sampled value with a resampled one. The following lemma is by Barthe et al. [27] with the informal notation we used in the proof obligation above, as introducing their notation would unnecessary require additional sections.

Lemma 5.1.1 (Lazy/eager sampling). *Let $C[\bullet]$ be a context, c_1 and c_2 commands, E a boolean expression, δ a distribution expression, and v a variable such that $C[\bullet]$ does not modify $FV\,(e) \cup FV\,(\delta)$[1] and does not use v. Assume*

$$\begin{aligned} \Psi \wedge e \Rightarrow & \\ & v \xleftarrow{\$} \delta; c_1; \textbf{if } e \textbf{ then } v \xleftarrow{\$} \delta; \;\sim\; v \xleftarrow{\$} \delta; c_1; \\ & \Rightarrow M_1.v \simeq M_2.v, \end{aligned} \tag{5.2}$$

where Ψ states that the values used in c_1 and c_2 are equivalent, and

$$\begin{aligned} \Psi \wedge \neg e \Rightarrow & \\ & c_2; \;\sim\; c_2; \\ & \Rightarrow M_1.v \simeq M_2.v \wedge \neg e. \end{aligned} \tag{5.3}$$

[1] FV are the free variables in an expression.

Let $c = \textbf{if } e \textbf{ then } v \xleftarrow{\$} \delta; c1; \textbf{ else } c2;$ *and* $c' = \textbf{if } e \textbf{ then } c1; \textbf{ else } c2;$*, then*

$$\begin{aligned} &\Psi \wedge e \Rightarrow \\ &\qquad C[c]; \textbf{if } e \textbf{ then } v \xleftarrow{\$} \delta; \quad \sim \quad v \xleftarrow{\$} \delta; C[c']; \\ &\Rightarrow M_1.v \simeq M_2.v. \end{aligned} \tag{5.4}$$

The intuitive discussion that they give is the following.

> In the above lemma e indicates whether v has not been used in the game since it was last sampled. If it has not been used, then it is perfectly fine to resample it. The first two hypotheses ensure that e has exactly this meaning, c_1 must set it to `false` if it has used the value sampled in v, and c_2 must not reset e if it is `false`. The first hypothesis is the one that allows to swap c_1 with $v \xleftarrow{\$} \delta$, provided the value of v is not used in c_1. Note that, for clarity, we have omitted environments in the above lemma, and so the second hypothesis is not as trivial as it may seem because both programs may have different environments.

We stress that the context $C[\bullet]$ and the environments in the above lemma are already complex in the task of proving the indistiguishability between a lazy construction and its corresponding eager construction, and they purposely omit further details. They would become even more complex in the case of directly proving indistiguishability between two lazy constructions like M_1 and M_2. This extra complexity is usually tamed by creating additional intermediate games.

From the Lemma 5.1.1 and the related discussion, we can highlight that using the eager-lazy approach to prove $M_1 \overset{c}{\equiv} M_2$ has the following limitations or problems.

First, an intermediate game Eager_i for each M_1 and M_2 needs to be created, that include the *resampling* operation; in fact, it is not present in either M_1 or M_2. So, from one indistiguishability to prove, $M_1 \overset{c}{\equiv} M_2$, we now have (at least) three[2], $M_1 \overset{c}{\equiv} \mathsf{Eager}_1$, $\mathsf{Eager}_1 \overset{c}{\equiv} \mathsf{Eager}_2$, and $\mathsf{Eager}_2 \overset{c}{\equiv} M_2$.

Second, two restrictions or assumptions need to be valid: i) the space of the sampled values must be finite, otherwise the an eager construction will never possibly terminate; and ii) especially when complexity is to be considered, the whole eager process has to be bound, for example by a polynomial, not to jeopardise the final result.

And third, and perhaps the most challenging, if a value is sampled in one of the algorithms and other variables are *dependent* on it, the former approach needs to re-sample that value, and the new value should affects in *cascade* all the dependent variables. The side effects of its value may modify the behaviour of other procedures in such a way that the Lemma 5.1.1 may not be even applicable.

[2]The equivalence of the lazy construction and eager construction must be proved separately using that lemma.

Contribution

With this work, we propose an alternative proof strategy, illustrated in Figure 5.2, that addresses the problems of the eager-lazy approach by introducing basic information flow support to the algorithmic language.

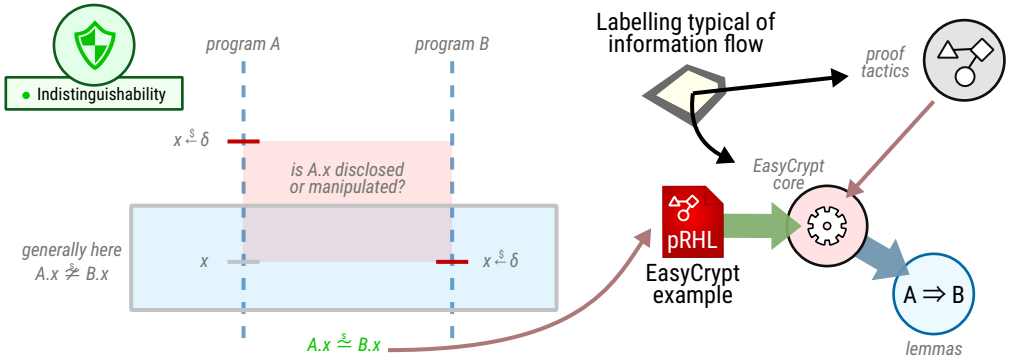


Fig. 5.2 Our proposed alternative; without our extension, EasyCrypt could reason only if the random samplings appear in the same function/algorithm or in some cases with a very difficult strategy that require extra constructions.

The core idea of our proof strategy is to *embed* the knowledge of when a variable is used in the variable itself, rather than delegating this to logical propositions in the context or the environment. As a case study, we prove the equality of two constructions with multiple procedures where a random sampling of a value migrates from a procedure to another, and *both* procedures can be called by the adversary; nevertheless, our example is simply a general pattern that highlights the issue that is present in the same form in many indistinguishability proofs. To the best of our knowledge, this is the first time that such a theoretical proof for indistinguishability has been implemented for imperative code in a theorem prover. To do that, we extend the pWhile language, which is at the core of the pRHL reasoning in EasyCrypt, then we implement the new semantics spread across three proof tactics, that allow for reasoning about a new dedicated information flow type that label values conveniently.

5.2 Information flow in formal methods

We refer to the literature to highlight the difference in the context or in the formal settings that highlight the novelty of our implementation.

Theorem provers have been used in the literature to certify verified properties of programs with information flow [57, 122, 23], other approaches are language-based and aim to reach non-interference [165], some of which aim to produce code along with proofs [156]. Information flow is often studied along access control policies [155]; therefore, their relation to network security is stronger than their relation to cryptography. In our context, *information flow* is the analysis to track the

transfer of information from a variable to another during the flow of an execution of an algorithm. We implement the classic information flow technique of marking variables with security labels that can change during the flow.

The closest literature to our work is by Barthe et al. [24] and by Grimm et al. [92]. Barthe et al. build on top of the ideas of Swamy et al. [156] and Nanevski et al. [130] and extend refinement types to relational formulae. Differently from our approach, they express the non-interference properties in the post condition, rather than directly label the data with security flags, as it is usually done in information flow analysis. They offer a classic implementation of a random oracle, that lazily samples (and memoizes) the random function, using a mutable reference holding a table mapping hash queries. When relating two programs, the queries are made in both leftmost and rightmost executions. To verify the equivalence properties, they introduce several invariants that must hold and that relate the tables stored by either of the two executions. They also relate sampled entries with an injective function that ensures they have indistinguishable distributions [3]. This approach requires the *manual* insertion (and proof) of program-dependent lemmas, which is not the case for us.

The other work that relate to us is from Grimm et al. They provide logical relation for a state monad and use it to prove contextual equivalences. They show perfect security of one-time-pad, that is very basic and certainly can be described with the pRHL. They are able to automatically produce proofs based on the equivalence of (bijective) random sampling operations. However, their work do not apply to equivalences with oracles, that is our case study.

5.3 Preliminaries

5.3.1 Reasoning in the pRHL

We implemented our approach in the probabilistic relational Hoare logic (pRHL) with EasyCrypt. Both have been introduced in Section 2.5, and this section highlights the parts relevant to the discussion of the information flow labelling that we implemented. More information about EasyCrypt and the syntax of its language can also be found in [21, 3].

Algorithms in EasyCrypt lie inside modules, that are containers of global variables and procedures. Procedures capture the idea of algorithms running in a memory as an execution environment, and one can reason about their deterministic behaviour (HL) or probabilistic behaviour (pHL) expressing the outcome of such procedures. For the most, we report the notation for the semantics of the pRHL judgements from [25]. An example of an HL and a pHL judgements where running a procedure c for every memory m with precondition Ψ and post-condition Φ are denoted as

$$\models c : \Psi \Rightarrow \Phi \qquad \text{HL judgement}$$
$$\models c : \Psi \Rightarrow \Phi < p \qquad \text{pHL judgement}$$

[3] This is what is usually done in the pRHL.

where in both we reason about Φ after running c, assuming that Ψ held true before running c, with the difference that, while in the first we reason directly about Φ, in the second we reason about the probability of the event modelled by Φ to be the specified relation with the real number p (in the specific example *less-than* p). We use similar notation for the pRHL judgements, used to reason about two procedures in comparison. Given two procedures c_1 and c_2 for any memory they run, a precondition Ψ and a post-condition Φ, then we write

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \tag{5.5}$$

to say that if the precondition Ψ holds in the context of initial memories relating to c_1 and c_2, then the post-condition Φ holds true, relating to the memories modified accordingly with the execution of c_1 and c_2. Both the precondition and the post-condition can refer to variables in the memory and contain relations about them; additionally, the post-condition can refer to the return value of the procedures. From pRHL judgements in Equation 5.5, one can derive probability claims to prove security of cryptographic constructions. Basically, we want to relate the pRHL judgement to two events, E_1 and E_2, that refer to the memories $\mathcal{M}_1$ and $\mathcal{M}_2$ in which c_1 and c_2 respectively run. If the judgement is valid and $\Phi \Rightarrow E_1 \Rightarrow E_2$, then the judgement is interpreted as an inequality between probabilities, if the precondition Ψ holds for every initial memories $\mathcal{M}_1$ and $\mathcal{M}_2$, denoted as $\mathcal{M}_1 \Psi \mathcal{M}_2$. Formally,

$$\begin{aligned} &(\models c_1 \sim c_2 : \Psi \Rightarrow \Phi) \wedge (\mathcal{M}_1 \Psi \mathcal{M}_2) \wedge (\Phi \Rightarrow E_1 \Rightarrow E_2) \\ \Rightarrow\ &\mathsf{Pr}\left[c_1 \left\langle \mathcal{M}_1 \right\rangle : E_1\right] \leq \mathsf{Pr}\left[c_2 \left\langle \mathcal{M}_2 \right\rangle : E_2\right]. \end{aligned}$$

In the case when c_1 or c_2 return a value, the events E_1 and E_2 may involve the return value, which we generically denote as $r \left\langle \mathcal{M}_1 \right\rangle$ and $r \left\langle \mathcal{M}_2 \right\rangle$ for respectively side 1 and side 2 of the judgement. <

5.4 The pifWhile language

The first step of our contribution is the modification of the core language of EasyCrypt to smoothly work with labelled values.

The imperative code at the core of EasyCrypt follows the syntax of the pWhile language [25], and its semantics is determined by the proof tactics of the theorem prover. A program in the pWhile language is defined as the following set of commands:

$C ::=$	$\mathbf{skip}$	no operation
$\mid$	$V \leftarrow E$	deterministic assignment
$\mid$	$V \xleftarrow{\$} E_\Delta$	probabilistic assignment
$\mid$	$C; C$	sequence
$\mid$	$\mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C$	conditional branch
$\mid$	$\mathbf{while}\ E\ \mathbf{do}\ C$	while loop

where V are variables bound in the memory of C, E is an expression, E_Δ is an expression of type distribution.

Information flow labelling may be implemented to transparently extend the semantics of the already existing syntax; however, we opted for extending the syntax with two dedicated statements, *secure assignment* and *secure probabilistic assignment*, that manipulate variables labelled for information flow control purposes. The reason of our choice is twofold. First, we do not affect the semantics of the other statements, therefore the soundness of all the theories across the theorem prover cannot be jeopardised by our extension. Second, the code produced is as clear as before, the usage of the assignment from and sampling to the special variables is very transparent and mistake-free. In fact, the two new statements behave as the regular deterministic and probabilistic assignments, apart from their special semantics during the proofs. In particular, we extended the pWhile language with two syntax symbols, $\hookleftarrow$ for *secure assignment* and $\overset{\$}{\hookleftarrow}$ for *secure probabilistic assignment*, to the pifWhile language. A program in the pifWhile language is defined as the following set of commands:

$\widetilde{C} ::=$	C	any command in the pWhile
	$\mid \widetilde{C}; \widetilde{C}$	sequence
	$\mid \textbf{if } E \textbf{ then } \widetilde{C} \textbf{ else } \widetilde{C}$	conditional branch
	$\mid \textbf{while } E \textbf{ do } \widetilde{C}$	while loop
	$\mid V \hookleftarrow \widetilde{E}$	deterministic assignment from a labelled value
	$\mid \widetilde{V} \overset{\$}{\hookleftarrow} E_{\Delta}$	probabilistic assignment to a labelled value

where $\widetilde{E}$ and $\widetilde{V}$ are respectively an expression and a variable labelled for information flow purposes, and the two syntax rules directly related to the added symbols are framed. Our extensions are the last two statements whose semantics will be explained in Section 5.6. In summary, the right hand value of the secure assignment is to be treated with the information flow labelling that we implemented, as well as the left hand value of the secure probabilistic assignment.

5.5 Information flow support

The most common practice to extend a language with information flow theory is through associating security labels to variables. Some implementations delegate all the information flow labels to the language interpreter [23], while we decided to make it explicit in the language itself by creating a new type that is a triplet. In particular, we associate them their value, the distribution from where they are sampled (if any), and a *confidentiality* label. For the sake of our demonstration, we did implement basic all-or-nothing confidentiality, that is the value can be labelled as either secure or leaked, leaving partial leakage to future improvement of our work.

Delegating labels to the language interpreter has the benefit of managing labels transparently to the programmer, who is not required to know how to manipulate them. However, since information flow analysis can be an expensive task, this approach still requires extra syntax for letting the coder to decide which variables must be treated or not in the analysis. On the contrary, a great advantage of our approach,

explicit to the language, is that the mathematical theory of the the information flow labelling can be extended without the need for changing the core of the language.

We now introduce the definitions and then notation of the security labels that we associate to the variables. We denote the family of all the distributions over any set by Δ, and the set of all the distributions over a generic set X by Δ_X. Since we introduce labelling for merely controlling information leakage, we introduce a new EasyCrypt theory named `Leakable`. Here we define two new types, *confidentiality* and *leakable*. The former type models a set $\mathcal{C}$ with only two values, H interpreted as *secret* and L interpreted as *leaked* to the adversary.

$$\mathcal{C} \equiv \{\mathrm{H}, \mathrm{L}\}\,.$$

The latter type models a family of sets that relate to a generic set X whose elements are labelled with a distribution over X and a confidentiality value:

$$\widetilde{X} \equiv X \times (\Delta_X \cup \{\bot\}) \times \mathcal{C}.$$

where $\bot$ is used if the value is not associated to a sampling distribution. Unions with the $\bot$ value can be easily implemented through option (or maybe) type. Due to the nature of the leakable type as a triplet, projection functions are already defined in the language. For any $\widetilde{x} = (x, \delta, c) \in \widetilde{X}$, we have the following functions:

$$\pi_1\left(\widetilde{x}\right) \stackrel{\text{def}}{=} x, \quad \pi_2\left(\widetilde{x}\right) \stackrel{\text{def}}{=} \delta, \quad \pi_3\left(\widetilde{x}\right) \stackrel{\text{def}}{=} c.$$

We define three additional functions over leakable values: (i) $\Lambda : \widetilde{X} \to \{\mathtt{T}, \mathtt{F}\}$ testing whether a leakable value has been leaked or not, (ii) $\in_{\mathrm{R}}\colon \widetilde{X} \to \Delta_X \to \{\mathtt{T}, \mathtt{F}\}$ whose output is `T` if the leakable value is sampled from the provided distribution and `F` otherwise, and (iii) $\simeq\colon \widetilde{X} \to \widetilde{X} \to \{\mathtt{T}, \mathtt{F}\}$ modelling the equality of two leakable values ignoring the confidentiality label. So, let $\widetilde{v} = (v, \delta_v, c_v)$ and $\widetilde{w} = (w, \delta_w, c_w)$ be two leakable values over the set X ($\widetilde{v}, \widetilde{w} \in \widetilde{X}$) and $\delta \in \Delta_X$ be a distribution over the same set, then we define

$$\begin{aligned} \Lambda\,\widetilde{v} &\stackrel{\text{def}}{=} c_v \neq \mathrm{H}, \\ \widetilde{v} \in_{\mathrm{R}} \delta &\stackrel{\text{def}}{=} \delta_v = \delta, \\ \widetilde{v} \simeq \widetilde{w} &\stackrel{\text{def}}{=} v = w \wedge \delta_v = \delta_w. \end{aligned}$$

An extract of the code implementing the feature above described is in Figure 5.3.

5.6 Proof tactics

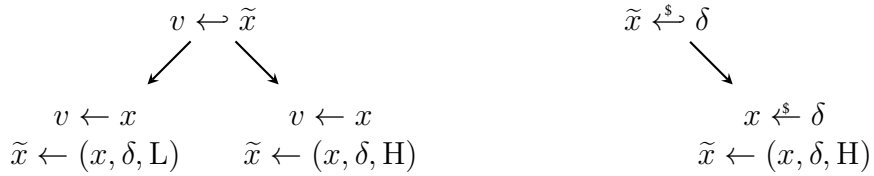
Tactics are inference rules allowing (part of) a theorem, or goal, to mutate into another goal. The proof is not completed until a final tactic is able to derive the tautology from the current theorem. Sometimes the goal structure can be split into

Fig. 5.3 Implementation of the information flow types, predicates, and operations

```
type confidentiality = [ SECRET | LEAKED ].
type 'a leakable = 'a * ('a distr) option * confidentiality.

op is_secret ['a] (v: 'a leakable) = SECRET = v.`3.
op is_leaked ['a] (v: 'a leakable) = !(is_secret v).
op sampled_from ['a] (d: 'a distr) (v: 'a leakable) = v.`2 = Some d.

op ovd_eq ['a] (v w: ('a leakable) option) =
  ((oget v).`1, (oget v).`2) = ((oget w).`1, (oget w).`2).

abbrev (===) ['a] (v w: ('a leakable) option) = ovd_eq v w.
abbrev (<=)  ['a] (v: 'a leakable) (d: 'a distr) = sampled_from d v.
```

several sub-goals to be independently proven, then finally combined altogether. In the pRHL, the proof can be seen as an execution environment of an algorithm, and the tactics confer semantics to the syntax used in the code of the algorithm.

Before introducing the tactics to manipulate the new syntax ($\hookleftarrow$ and $\overset{\$}{\hookleftarrow}$), we illustrate their semantics, using the notation introduced in Sections 2.5 and 5.3. Roughly, $\hookleftarrow$ and $\overset{\$}{\hookleftarrow}$ are syntactic sugar for the deterministic assignment $\leftarrow$ and probabilistic assignment $\overset{\$}{\leftarrow}$ respectively, with the side effect of updating the information flow labels, depending on the context they are manipulated.



We describe the three tactics related to the syntax introduced in Section 5.4, for the special types introduced in Section 5.5:

declassify makes a controlled variable be labelled as leaked, L;

secrnd makes a controlled variable in a probabilistic assignment from a distribution δ be labelled as sampled from δ and secret, H.

secrndasgn works when two procedures are in relation and makes a probabilistic assignment mutate to a simple assignment.

The tactics *declassify* and *secrnd* work in the HL and pHL similarly, and in the pRHL can be called side by side, their semantics is summarised in Figure 5.4 (we write the semantic rules where the top part are the conditions that need to be for the tactic to be applied, so the logical implication is bottom-up). The tactic declassify mutates the syntax of $\hookleftarrow$ into two deterministic assignments: the first labels the right hand value of $\hookleftarrow$ as leaked, and the second assigns the bare value of the labelled variable to the left hand value of $\hookleftarrow$. The tactic secrnd mutates the syntax of $\overset{\$}{\hookleftarrow}$ into a probabilistic

assignment and a deterministic assignment: the first binds a new[4] variable v in the memory $\mathcal{M}$ where the algorithm runs, then it assigns to v a value sampled from the distribution δ at the right hand of $\overset{\$}{\hookleftarrow}$, and the second assigns to the left hand the value v labelled with δ and H (secret). We remark that at this point of the flow, freshly sampled values must be secret; whether it will remain secret until the end of the flow, it depends on the other parameters of the goal, precondition, postcondition and the successive statements. As the two tactics declassify and secrnd simply rewrite the newly introduced syntax to assignments that are already supported in EasyCrypt, their soundness is entailed by that of the language.

Fig. 5.4 Proof rules in the HL for *declassify* and *secrnd*. Their corresponding to pHL and sided pRHL are trivially constructed from them.

$$\text{[declassify]}\frac{r : Y \qquad m : \widetilde{Y} \qquad \models c; r \hookleftarrow m : \Psi \Rightarrow \Phi}{\models c; m \leftarrow (\pi_1(m), \pi_2(m), \mathrm{L}); r \leftarrow \pi_1(m) : \Psi \Rightarrow \Phi}$$

$$\text{[secrnd]}\frac{m : \widetilde{Y} \qquad v : Y \qquad \models c; m \overset{\$}{\hookleftarrow} \delta : \Psi \Rightarrow \Phi \qquad v \notin FV(\mathcal{M})}{\models c; v \overset{\$}{\hookleftarrow} \delta; m \leftarrow (v, \delta, \mathrm{H}) : \Psi \Rightarrow \Phi}$$

The semantics for the tactic secrndasgn, illustrated in Figure 5.5, is more complex and involves an invariant that must hold before and after calling every corresponding procedures of the two construction. For a variable, the invariant can be as simple as the following

$$\overset{\hookleftarrow}{I}(\widetilde{v}, \widetilde{w}, \delta) \overset{\text{def}}{=} \neg \Lambda\, \widetilde{w} \ \wedge\ \widetilde{w} \in_{\mathrm{R}} \delta.$$

When modelling a random oracle, the values are usually stored in a map or a table. We implemented support to maps that model functions, where initially all the domain is mapped to $\perp$ and it is interpreted as an empty map. A map M shaping a partial function from X to Y is defined to always reach an option codomain, $M : X \to Y \cup \{\perp\}$. We use the notation $M(x)$ to denote the value of the domain element x in the map M. The domain of definition of the map M, denoted as M_X, is defined as

$$M_X \overset{\text{def}}{=} \{x \in X | M(x) \neq \perp\}.$$

So an *empty map* is easily defined as the empty domain of definition of the map itself.

$$M = \emptyset \Leftrightarrow M_X = \emptyset.$$

When the same tactics are applied to maps, the invariant looks more complicated, because the maps would not be consistent if compared directly. So more properties need to be taken into account. Formally, given two maps $M, N : X \to \widetilde{Y} \cup \{\perp\}$ and a distribution $\delta \in \Delta_Y$ over the set Y, we define the invariant for secure random

[4]The variable has never been declared or used in the memory before this point in the flow.

assignment from the map N to the map M as:

$$\begin{aligned}\overset{\hookleftarrow}{I}\,(M,N,\delta) \overset{\text{def}}{=} \forall x \in X,\\ &(x \in N_X \Rightarrow N\,(x) \in_{\mathrm{R}} \delta)\\ \wedge\ &(x \in M_X \Rightarrow x \in N_X \wedge \pi_1\,(M\,(x)) = \pi_1\,(N\,(x)))\\ \wedge\ &(x \in M_X \Rightarrow \Lambda\, M\,(x) \Rightarrow M\,(x) = N\,(x))\\ \wedge\ &(x \notin M_X \Rightarrow x \in N_X \Rightarrow \neg\, \Lambda\, N\,(x))\,.\end{aligned}$$

The invariant ensures that for all elements x in the domain X the following properties hold: (i) if x is set in N, then it is distributed as δ; (ii) if x is set in M, then it is also set in N and both hold the same sampled value; (iii) if x is set in M and it has been leaked to the adversary, then x is also set in N and its image equal that in N, $M\,(x) = N\,(x)$; and (iv) if x is not set in M but is set in N, then the value (in N) is secret.

The proof case supported by the tactic secrndasgn is in a relational proof, where the procedure at left shows a sampling from the distribution δ to a labelled value v, followed by an assignment from it, and the procedure at right shows the same assignment but without any sampling:

$$v \overset{\$}{\leftarrow} \delta; x \leftarrow v \quad \sim \quad x \leftarrow v$$

The only case when these two algorithms behave the same is when v in the right algorithm is *secret* and its distribution label is δ. In fact, if a value has been sampled in the past but remains secret, no output from any other function may have disclosed its content, therefore we can mutate the sampling in the left algorithm with an assignment whose right value is the one in v from the rightmost algorithm. As shown in Figure 5.5, the tactics splits the current in two sub-goals.

Fig. 5.5 Proof rule in the pRHL for *secrndasgn*. This tactic requires to prove two sub-goals.

$$\text{[secrndasgn-g1]}\ \frac{\begin{array}{c} m\,\langle \mathcal{M}_1\rangle, m\,\langle \mathcal{M}_2\rangle : X \to \widetilde{Y} \cup \{\bot\} \qquad x : X \qquad \widetilde{v} : \widetilde{Y} \qquad \delta \in \Delta_Y \\ s_1 = m\,(x) \overset{\$}{\hookleftarrow} \delta \qquad a_1 = r \hookleftarrow m\,(x) \qquad a_2 = r \hookleftarrow m\,(x) \\ \models c_1; s_1; a_1 \sim c_2; a_2 : \Psi \Rightarrow \Phi \qquad \mathcal{M}_1' = \mathcal{M}_1 \text{ with } \widetilde{v} \end{array}}{\begin{array}{l} \models c_1\,\langle \mathcal{M}_1'\rangle \sim c_2 : \\ \Psi \wedge \widetilde{v}\,\langle \mathcal{M}_1'\rangle = m\,\langle \mathcal{M}_2\rangle \Rightarrow \Phi \wedge \widetilde{v}\,\langle \mathcal{M}_1'\rangle = m\,\langle \mathcal{M}_2\rangle \wedge \neg\, \Lambda\, \widetilde{v}\,\langle \mathcal{M}_1\rangle \end{array}}$$

$$\text{[secrndasgn-g2]}\ \frac{\begin{array}{c} m\,\langle \mathcal{M}_1\rangle, m\,\langle \mathcal{M}_2\rangle : X \to \widetilde{Y} \cup \{\bot\} \qquad x : X \qquad \widetilde{v} : \widetilde{Y} \qquad \delta \in \Delta_Y \\ s_1 = m\,(x) \overset{\$}{\hookleftarrow} \delta \qquad a_1 = r \hookleftarrow m\,(x) \qquad a_2 = r \hookleftarrow m\,(x) \\ \models c_1; s_1; a_1 \sim c_2; a_2 : \Psi \Rightarrow \Phi \qquad \mathcal{M}_1' = \mathcal{M}_1 \text{ with } \widetilde{v} \end{array}}{\begin{array}{l} \models \{c_1; m\,(x) \leftarrow \widetilde{v}; a_1\}\,\langle \mathcal{M}_1'\rangle \sim c_2; a_2 : \Psi \wedge \widetilde{v}\,\langle \mathcal{M}_1'\rangle = m\,\langle \mathcal{M}_2\rangle \\ \qquad \Rightarrow \Phi \wedge m\,(x)\,\langle \mathcal{M}_1'\rangle \in_{\mathrm{R}} \delta \wedge \overset{\hookleftarrow}{I}\,(m\,\langle \mathcal{M}_1'\rangle, m\,\langle \mathcal{M}_2\rangle, \delta) \end{array}}$$

The first goal requires that the value $\widetilde{v}$ has not been leaked before the execution of the relevant statements:

$$\neg \Lambda\, \widetilde{v}\, \langle \mathcal{M}_1 \rangle\,.$$

The second goal mutates the probabilistic assignment into a deterministic assignment with the same exact value in the other construction. Importantly, this detail solves the *cascading* problem of re-sampling that affects the eager-lazy approach. Borrowing the stored value from the other construction guarantees that no further assignments *need* to be adjusted where the stored value has been used[5]. As we will show in Section 5.7, the invariant will play a crucial role to support the two proof obligations generated when applying the tactic secrndasgn.

Solving this problem will allow us to relate the *already sampled value* from the other construction avoiding the need for re-sampling: we emphasise that this tactic is the key to overcome the cascading adjustment that may be required if using the *eager* tactics involving re-sampling.

As a final comment, we notice that the tactic secrndasgn is not *symmetric*, i.e. it requires the probabilistic assignment $\xleftarrow{\$}$ to appear on the leftmost algorithm. However, the logic for pRHL judgements allows to swap the leftmost algorithm with the rightmost (equivalence of algorithms is symmetric). This can be easily done with the EasyCrypt tactic symmetry. If the probabilistic assignment $\xleftarrow{\$}$ appears on the rightmost algorithm, then symmetry can be simply applied before secrndasgn. So, extending our tactic to be symmetrically applicable would be a very minor improvement and definitely is not a limitation in its logic.

5.6.1 About the soundness of introduced tactics

Toward soundness, it is of crucial importance[6] to forbid the coder to abuse our syntax by modifying incorrectly the labels in the construction itself. If incorrectly labelled, in fact, the tactics introduced would make it possible to disprove the tautology, i.e. $\mathtt{T} = \mathtt{F}$. This would be a very unpleasant situation as the automatic SMT solvers in EasyCrypt may find their way to prove everything, as implied from the assumption F. To avoid such situations, our implementation forbids direct or indirect usage (inside other expressions) of variables and maps with information flow labels if not with their dedicated syntax; moreover, no regular tactics are able to discharge statements with labelled values.

The syntactic structures $\hookleftarrow$ and $\xleftarrow{\$}$ that we introduced are simply a translation to multiple assignments, and the tactics that manipulate them in the proof environment can be seen more as a semantic translation rather than a full language extension. As introduced in Section 5.6, the two tactics declassify and secrnd, illustrated in Figure 5.4, simply rewrite the newly introduced syntax to assignments that are already supported in EasyCrypt. Their soundness is trivially entailed by the language itself: without the tactic secrndasgn, the syntactic structures $\hookleftarrow$ and $\xleftarrow{\$}$ are uniquely mapped to multiple statements with no other effects whatsoever:

[5]Using that value does not automatically imply that it has been disclosed to the adversary.

[6]I explicitly thank Benjamin Gregoire for this important note.

$$\begin{array}{c} v \hookleftarrow \widetilde{x} \\ \Big\downarrow \text{declassify} \\ v \leftarrow x \\ \widetilde{x} \leftarrow (x, \delta, \mathrm{L}) \end{array} \qquad \begin{array}{c} \widetilde{x} \overset{\$}{\hookleftarrow} \delta \\ \Big\downarrow \text{secrnd} \\ x \overset{\$}{\leftarrow} \delta \\ \widetilde{x} \leftarrow (x, \delta, \mathrm{H}) \end{array}$$

It is important to remark that we modified the parser of the language such that it forbids regular assignment from and to labelled type, e.g. $\widetilde{x} \leftarrow (x, \delta, \mathrm{L})$ or $x \leftarrow f(\widetilde{x})$ for any f. This way, the user cannot freely access the internal value nor manipulate the confidentiality label, but needs to use their dedicated syntax.

Conversely, the introduction of the tactic secrndasgn requires more attention, as it introduces ambiguity of their translation: in particular, it is able to mutate a probabilistic assignment into a deterministic assignment.

$$\begin{array}{c} \widetilde{y} \overset{\$}{\hookleftarrow} \delta \\ \Big\downarrow \text{past flow} \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \begin{array}{ll} \widetilde{x} \overset{\$}{\hookleftarrow} \delta & \\ x \hookleftarrow \widetilde{x} & \sim \quad y \hookleftarrow \widetilde{y} \end{array} \\ \Big\downarrow \text{secrndasgn} \\ \begin{array}{ll} \widetilde{x} \leftarrow \widetilde{y} & \\ x \hookleftarrow \widetilde{x} & \sim \quad y \hookleftarrow \widetilde{y} \end{array} \end{array}$$

Nevertheless, the reasoning behind this semantic behaviour is already present in the EasyCrypt language and exactly the same as the tactic rnd when applied to probabilistic assignments drawing from the same distribution in the pRHL. The tactic rnd is well supported in EasyCrypt and illustrated in Figure 5.6.

Fig. 5.6 The tactic rnd in the pRHL in EasyCrypt can be applied to probabilistic assignments; logic implication is bottom-up. The two isomorphisms $f_{2,1}$ and $f_{1,2}$ can be explicitly provided.

$$\text{[rnd]}\ \frac{\begin{array}{c} x, y : X \qquad f : X \to X \to \{\mathtt{T}, \mathtt{F}\} \qquad \delta_1, \delta_2 \in \Delta_X \qquad f_{2,1}, f_{1,2} : X \to X \\ \models x \overset{\$}{\leftarrow} \delta_1 \sim y \overset{\$}{\leftarrow} \delta_2 : \Psi \Rightarrow f(x, y) \end{array}}{\begin{array}{l} \models \{\} \sim \{\} : \Psi \Rightarrow \\ \qquad \forall y, \mathsf{Pr}[y \leftarrow \delta_2] > 0 \Rightarrow y = f_{1,2}(f_{2,1}(y)) \\ \qquad \forall y, \mathsf{Pr}[y \leftarrow \delta_2] > 0 \Rightarrow \mathsf{Pr}[y \leftarrow \delta_2] = \mathsf{Pr}[f_{2,1}(y) \leftarrow \delta_2] \\ \qquad \forall x, \mathsf{Pr}[x \leftarrow \delta_1] > 0 \Rightarrow x = f_{2,1}(f_{1,2}(x)) \wedge f(x, f_{1,2}(x)) \end{array}}$$

The limitation of the tactic rnd is that it requires both probabilistic assignments to happen *at the bottom* of the flow of the proof: our tactic secrndasgn overcomes this limitation by applying the reasoning of rnd to two probabilistic assignments that happen to be in different parts of the flow (and therefore cannot possibly be at the bottom of the flow of the proof). We remark that our approach automatically applies the reasoning of rnd and therefore implements a very different idea from the eager-lazy approach. In the eager-lazy approach, in fact, a redundant probabilistic assignment is artificially injected into the code that *prepares* the algorithm to be suitable for rnd to be independently applied later. Differently, our approach directly applies the result of the rnd by making sure that the *missing* probabilistic assignment was actually executed previously in the flow and, importantly, its value has not been manipulated since the last time it was sampled.

Implicit application of the tactics rnd **and** swap

As the reader can see, the assignment $\widetilde{x} \leftarrow \widetilde{y}$ produced by secrndasgn captures their equality (after the execution of the assignment). Such equality can be exactly the post-condition of the tactic rnd illustrated in Figure 5.6, where $\widetilde{x} = (x, \delta_x, c_x)$, $\widetilde{y} = (y, \delta_y, c_y) \in \widetilde{X}$, the isomorphisms $f_{1,2}$ and $f_{2,1}$ are both the identity, $\delta = \delta_1 = \delta_2$ is a distribution over X, and f is the proposition $\widetilde{x} = \widetilde{y}$, under the precondition $\Psi = (\delta_x = \delta \wedge \delta_y = \delta \wedge c_x = c_y)$. With such constraints, the post-condition propositions produced by applying rnd are trivially true:

$$\text{[rnd]}\frac{\begin{array}{c}\widetilde{x}, \widetilde{y} : \widetilde{X} \qquad\qquad \delta \in \Delta_X \\ \models x \xleftarrow{\$} \delta \sim y \xleftarrow{\$} \delta : (\delta_x = \delta \wedge \delta_y = \delta \wedge c_x = c_y) \Rightarrow \widetilde{x} = \widetilde{y}\end{array}}{\begin{array}{l}\models \{\} \sim \{\} : (\delta_x = \delta \wedge \delta_y = \delta \wedge c_x = c_y) \Rightarrow \\ \qquad \forall y, \mathsf{Pr}\left[y \leftarrow \delta\right] > 0 \Rightarrow y = y \\ \qquad \forall y, \mathsf{Pr}\left[y \leftarrow \delta\right] > 0 \Rightarrow \mathsf{Pr}\left[y \leftarrow \delta\right] = \mathsf{Pr}\left[y \leftarrow \delta\right] \\ \qquad \forall x, \mathsf{Pr}\left[x \leftarrow \delta\right] > 0 \Rightarrow x = x \wedge \widetilde{x} = \widetilde{x}\end{array}}$$

The precondition $\Psi = (\delta_x = \delta \wedge \delta_y = \delta \wedge c_x = c_y)$ is forcibly introduced by the invariants introduced by the application of the tactic secrndasgn and must be proved, otherwise the proof cannot be completed. To complete the soundness in the reasoning of the tactic secrndasgn, we need to show that the above situation is uniquely and unequivocally captured. This can be done by showing that

- the past flow definitely contained a corresponding sampling (that could have been discharged with rnd), and
- that sampling could have been *moved* to the current point in the flow.

The first part is proved by the invariant that requires the value of $\widetilde{y}$ to be labelled as H and sampled from the distribution δ. This requirement is *uniquely* producible by the application of the tactic secrnd anywhere in the past flow: the tactic can be applied only to the syntax $\xleftarrow{\$}$, that unfolds exactly to $x \xleftarrow{\$} \delta$ and produces the statement $\widetilde{x} \leftarrow (x, \delta, H)$ that validate the precondition Ψ.

The last part can be seen as a virtual extension[7] of the tactic swap in EasyCrypt: indeed, swap is very simple and allows to *move* a statement from a position to another and it fails if the swapped statements are dependent. The only constructs that are able to manipulate (read or write) a labelled value are either $\overset{\$}{\hookleftarrow}$ or $\hookleftarrow$ that can be singularly discharged only by secrnd and declassify respectively, or in an equivalence by secrndasgn itself. We have two cases to show the independency since the **last sampling**: (i) the value $\widetilde{y}$ have never been used, (ii) the value $\widetilde{y}$ have been read. Write operations are not in our case list, as they would represent the *last sampling* operation.

The first case (i) is trivial, as there cannot be dependent lines and the reasoning is as sound as the tactic swap. The second and last case (ii) implies that the variable $\widetilde{y}$ has been accessed in another statement (assignments, or condition in guards or loops). The parser allows only syntax $\hookleftarrow$ to be used forbidding anything else. This way, the only tactic to discharge the special assignment $\hookleftarrow$ is declassify, whose effect will be to mark $\widetilde{y}$ as leaked, L: this will simply prevent the user to prove part of the post-condition of secrndasgn $-$ g1 (see Figure 5.5), where the variable is required to be secret, H, and the proof cannot be completed.

As we have covered all possible scenarios, this concludes our discussion about the soundness of the proposed approach and extension.

5.7 Sample usage in pRHL proofs

To show the effectiveness and simplicity of application of our approach, we show an example where an adversary cannot distinguish between two constructions, implemented as two modules M_1 and M_2, while given oracle access to a subset of the internal procedures of one module or (exclusively) the other. The proof structure reproduced here is exactly the same of many other indistinguishability proof where preconditions and post-condition are more complicated; this structure is common especially in simulation-based proofs of complex constructions: we used this approach in the proof of Sophos, see Chapter 6. In such and similar proofs a simulator replaces structures depending on (unknown) private inputs with randomly sampled values: in this Section, we show a very simplified case study.

Both the modules M_1 and M_2 emulate the behaviour of a random function $f : X \to Y$, as can be seen in Figure 5.7. Both implementations are lazy: the first time when the input $x \in X$ is used for calling the procedure f, a value y is freshly sampled at random from a distribution δ and stored to an internal map m, the next times when $f\,(x)$ will be called, the value will be retrieved its image from m. Additionally to f, the adversary can also call the procedure $g : X \to \{\perp\}$ whose signature is identical to f apart from the empty return value, i.e. it produces no output. The only difference between M_1 and M_2 is in the implementation of the function g, where in the former

[7] The tactic swap has the same limitation of rnd, i.e. it cannot reason outside of the current flow of execution.

Fig. 5.7 Sample equivalent programs.

```
module M1: RF = {
  var m: (X, Y leakable) fmap

  proc init() = { m = empty; }

  proc f(x: X) = {
    var ret: Y;

    if (!(dom m x)) {
      m.[x] </$ dv;
    }

    ret </ oget m.[x];

    return ret;
  }

  proc g(x: X) = { }
}.
```

```
module M2: RF = {
  var m: (X, Y leakable) fmap

  proc init() = { m = empty; }

  proc f(x: X) = {
    var ret;

    if (!(dom m x)) {
      m.[x] </$ dv;
    }

    ret </ oget m.[x];

    return ret;
  }

  proc g(x: X) = {
    if (!(dom m x)) {
      m.[x] </$ dv;
    }
  }
}.
```

the procedure g does nothing, and in the latter g actually fills the map (if not already) with a value that can be later retrieved by calling f on the same argument.

Formally, we can show that, given a non-singleton uniform distribution δ, and a probabilistic polynomial time adversary $\mathcal{A}$, then the advantage of distinguishing between the two constructions M_1 and M_2 is zero:

$$\mathbf{Adv}^{\mathcal{A}}_{M_1,M_2}\left(1^n\right) = 0.$$

Following the definition of advantage given in Section 2.1.3, the probability of distinguishing either must be the same

$$\mathsf{Pr}\left[\mathcal{A}\left(M_1\right) = 1\right] = \mathsf{Pr}\left[\mathcal{A}\left(M_2\right) = 1\right].$$

The proof of the above lemma follows the complete formal proof conducted in EasyCrypt, with the examples in Figure 5.7. Inspecting into the details of the proof of the above statement, the indistinguishability can be split into showing that the outputs of the corresponding procedures f in M_1 and M_2, upon the same inputs, are equally distributed. To keep consistency during the flow, we also use the invariant $\overleftrightarrow{I}\left(m\left\langle\mathcal{M}_2\right\rangle, m\left\langle\mathcal{M}_1\right\rangle, \delta\right)$. For simplicity, we denote $m\left\langle\mathcal{M}_1\right\rangle$ as M and $m\left\langle\mathcal{M}_2\right\rangle$ as N, so the invariant shows the same notation as defined in Section 5.6.

Formally, we have to prove the validity of the invariant before calling the distinguisher $\mathcal{A}$, along with the following pRHL judgements:

$$\models \{\} \sim \{\} : \Psi \Rightarrow \Phi \tag{5.6}$$
$$\models M_1.g\,(x) \sim M_2.g\,(x) : \Psi \Rightarrow \Phi \tag{5.7}$$
$$\models M_1.f\,(x) \sim M_2.f\,(x) : \Psi \Rightarrow \Phi \wedge r\,\langle \mathcal{M}_1 \rangle = r\,\langle \mathcal{M}_2 \rangle \tag{5.8}$$

where the precondition Ψ includes the invariant and the arguments passed to the function, $\Psi = \overleftrightarrow{I}\ (M, N, \delta) \wedge x\,\langle \mathcal{M}_1 \rangle = x\,\langle \mathcal{M}_2 \rangle$, the post-condition includes the invariant $\Phi = \overleftrightarrow{I}\ (M, N, \delta)$, and (the latest judgement only) includes the return values $r\,\langle \mathcal{M}_1 \rangle$ and $r\,\langle \mathcal{M}_2 \rangle$.

The first pRHL judgement (5.6) is trivial, as nothing is executed and Φ is obviously entailed by Ψ. The second judgement (5.7) shares similarities with the first, as $M_1.g\,(x) = \{\}$ is the empty algorithm and both produce no output. From here, we have two cases (a) when x is already in the map m of the memory $\mathcal{M}_2$, and (b), conversely, when $x \notin N_X$ and about to be sampled. In the case (a), we reduce to the first judgement (5.6), while in (b) we reduce to the following

$$\models \{\} \sim m\,(x) \xleftarrow{\$} \delta : \Psi \Rightarrow \Phi.$$

Here, the action of sampling into $m\,(x)\,\langle \mathcal{M}_2 \rangle$ does not jeopardise the validity of the invariant. In particular, only the last part of the invariant may be affected, i.e. when $x \notin M_X$ but $x \in N_X$. So, if we apply the tactic *secrnd*, then we can keep track of the confidentiality of $m\,(x)\,\langle \mathcal{M}_2 \rangle$ and prove that $\neg\,\Lambda\,m\,(x)\,\langle \mathcal{M}_2 \rangle$, as requested by the invariant. Finally, the third pRHL judgement (5.8) is the one that benefits the most from our work. All the cases when the value is in the domain of both M and N, i.e. $x \in M \wedge x \in N$ or $x \notin M \wedge x \notin N$, the algorithms of either sides are exactly the same, and doing consistent operations in both maps M and N does not affect the validity of the invariant. The most relevant and interesting part is, again, when $x \notin M_X$ but $x \in N_X$. Different from the judgement (5.7), the procedure also produces an output, that corresponds to the values stored in the memories: $m\,(x)\,\langle \mathcal{M}_1 \rangle$ and $m\,(x)\,\langle \mathcal{M}_2 \rangle$. In particular, we need to show that they are (probabilistically) the same, as in this case the judgement (5.8) reduces to the following:

$$\models m\,(x) \xleftarrow{\$} \delta; r \leftarrow m\,(x) \sim r \leftarrow m\,(x) : \\ \Psi \wedge x \notin M_X \wedge x \in N_X \Rightarrow \Phi \wedge r\,\langle \mathcal{M}_1 \rangle = r\,\langle \mathcal{M}_2 \rangle\,,$$

The information flow labels associated to the value $m\,(x)$ can be used to apply the secrndasgn tactic as explained in Section 5.6. Following the rules for the tactic as illustrated in Figure 5.5, the goal splits in two proof obligations:

$$\models \{\} \sim \{\} : \Psi' \Rightarrow \Psi' \wedge r\,\langle \mathcal{M}_1' \rangle = r\,\langle \mathcal{M}_2 \rangle \tag{5.9}$$
$$\models m\,(x) \leftarrow v; r \leftarrow m\,(x) \sim r \leftarrow m\,(x) : \Psi' \Rightarrow \Phi' \tag{5.10}$$

where

$$\begin{aligned}
&\mathcal{M}_1' = \mathcal{M}_1 \text{ augmented with } \widetilde{v},\\
&\Psi' = \Psi \wedge x \notin M_X \wedge x \in N_X \wedge v \left\langle \mathcal{M}_1' \right\rangle \simeq m\left(x\right) \left\langle \mathcal{M}_2 \right\rangle, \text{ and}\\
&\Phi' = \Phi \wedge r \left\langle \mathcal{M}_1' \right\rangle = r \left\langle \mathcal{M}_2 \right\rangle \wedge m\left(x\right) \left\langle \mathcal{M}_1' \right\rangle \in_{\mathrm{R}} \delta.
\end{aligned}$$

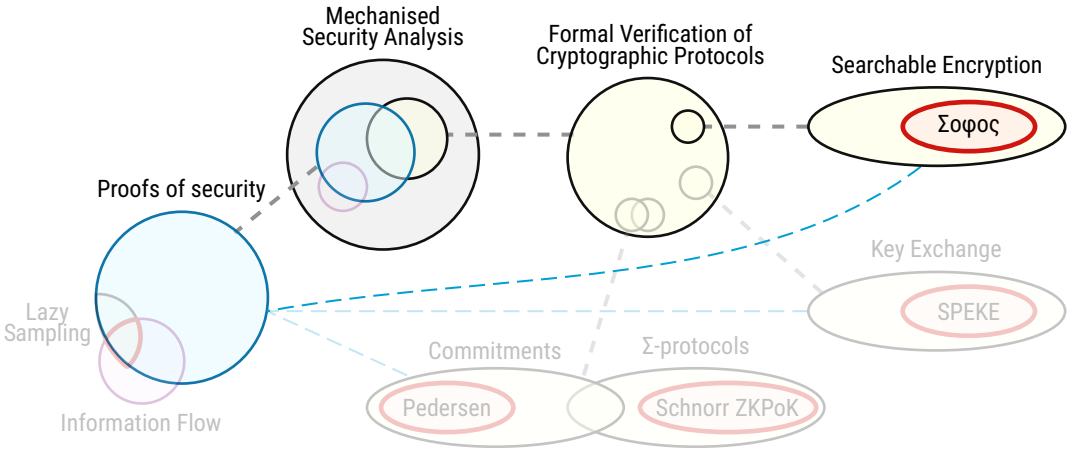
To finish the proof, we see that the proof obligation (5.9) holds, as the confidentiality of the value in $\widetilde{v}$ is secret as the invariant; this fact holds because the value *borrowed* from the map was sampled in the past, and it has never been used or revealed. The validity of the last proof obligation (5.10) is trivial, as the return values now are simple assignments of the same values in both sides, and the value clearly must have been labelled as sampled from δ (no other probabilistic assignments are even used in the structure of M_1 nor M_2).

5.8 Conclusion

In this chapter, we showed our implementation that uses information flow labelling with the purpose of proving indistiguishability between two constructions where random samplings are drawn in different procedures that are callable as oracles. Firstly, we extended the core syntax of EasyCrypt to label variables enabling for information flow analysis for code-based pRHL reasoning. Secondly, we implemented the semantics of such new syntax through creating new tactics to carry out proofs manipulating labelled variables and maps. We finally showed a case study that, if compared with the eager-lazy approach, can noticeably simplify the proof and does not involve additional intermediate games. Moreover, we see our contribution as a sound basis to support information flow analysis for code-based pRHL reasoning.

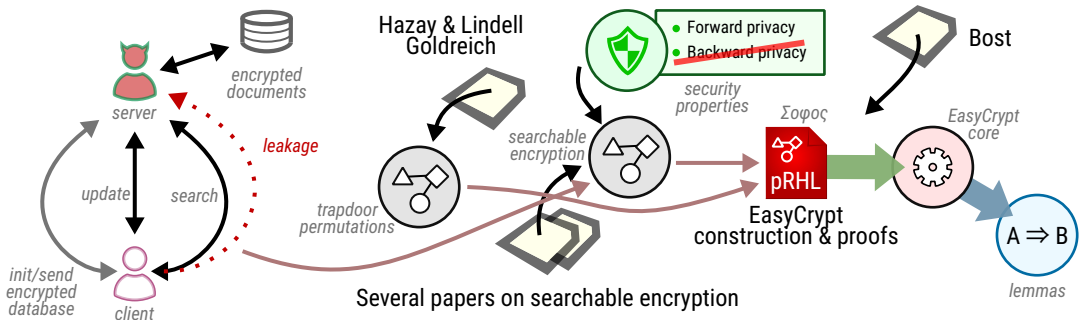
Chapter 6

Searchable encryption security in the computational model



In this chapter, we propose formal definitions for dynamic Symmetric Searchable Encryption (SSE) schemes, then we show a formal proof of forward secrecy with the aid of a mechanised formal proof in EasyCrypt for the recent SSE scheme Sophos, as a case study. The scheme inherently requires the proof to be carried out against an adaptive adversary and in the Random Oracle Model. As such, our mechanisation is the first of its kind for searchable encryption schemes. Furthermore, our work patches the on-paper proof, of which flaws may affect the proofs of successive proposed variants [54] that are based on the same simulation-based proof idea. Figure 6.1 illustrates our contribution, where we first propose generic definitions and support for SSE schemes to be able to describe different important SSE protocols. Translated to practice, we extended the corpus of theories of EasyCrypt to support abstract searchable encryption schemes, permutations, collection of one-way trapdoor permutations, function self-composition, as well as extending existing theories (in EasyCrypt) with novel proofs to aid our main proof. Then, we show the effectiveness of our mechanisation by implementing the real protocol $\Sigma o\varphi o\varsigma$ and proving forward secrecy.

Fig. 6.1 Description of our contribution for searchable encryption schemes.



6.1 Searchable Encryption

Day by day, an increasingly amount of sensible data is being outsourced to remote cloud servers by individuals, companies, and governmental bodies. Additional functionality over those data is offered in the form of cloud services such as email and collaborative tools, personal data synchronisation and backup, document sharing and file hosting, and other services. Searchable encryption is one of such services enabling a client to search documents by keyword, where the documents are encrypted and stored on an **untrusted** server. The security goal is to guarantee privacy to client's both documents and queries.

To achieve privacy, the simple basis is that of encrypting the data before sending it to the server. However, if a client wants the server to be able to perform operations over the encrypted data, then expensive cryptographic techniques are required, such as fully-homomorphic encryption [83] or ORAM-based constructions [132]. Their inefficiency excludes them from being adopted in practical implementations. To improve the efficiency, either functionality is reduced or security is relaxed to allow some information leakage. Finding an acceptable balance of *security* and *performance* is very important towards the adoption of practical secure implementations. Companies need to use such secure implementations in accordance not only with the security demand of clients, but also with legal regulations by which they need to abide, such as the most recent European General Data Protection Regulation (GDPR) approved in 2016, where, for example, outsourcing non-encrypted either financial data or computation of medical data becomes a felony.

There are two flavours of searchable encryption schemes, *static* and *dynamic*. Static schemes have been proposed where the corpus of documents cannot be updated by adding or removing documents [72, 60, 61, 71, 113, 114] among others. This is quite restrictive, as any change in the database would require the client to re-create and encrypt the whole new database and reinitialise the whole settings; so, *dynamic* schemes have been proposed to include the update functionality [84, 159, 108, 108, 152, 52, 81, 141] among others. The current state of the art studies *dynamic* schemes, where the database can be updated, and the database of documents may be detached

from the database containing the index of the documents[1], index based dynamic searchable encryption.

Efficient searchable encryption schemes allow for a small amount of leakage to the server, as the server must distinguish between encrypted documents containing a keyword from those that do not. The impact of such leakage has been underestimated for long time. This is witnessed by many conceptual and practical attacks that would allow the server to reconstruct a large portion of client's data or queries [102, 117, 58, 164] from the leakage. The research efforts are towards minimising such leakage to protect client's privacy. Researchers propose new definitions and constructions to describe the security of searchable encryptions schemes to address the security flaws. One of the most important property in dynamic schemes is *forward security*. Forward security prevents the keyword searched in the past from being related to newly updated documents [71]. The work that best highlighted the importance of forward security was by Zhang et al. [164], in which they showed how devastating file-injection attacks could be for queries' privacy. They show a series of attacks that can be run on searchable encryption schemes where the server can learn that newly added documents match previous search queries[2]. With such information and the ability to inject as few file as *logarithmic* to the number of keywords, a server could reconstruct the content of past queries, hence break their privacy. Forward secure schemes do not leak such information to the server, making those attacks ineffective.

Before the file-injection attack, only few schemes were forward secure [60, 152] among the efficient ones supporting updates, as opposed to others that were not [84, 108, 107, 59, 133, 115, 58]. Forward security became one of the main properties of novel dynamic schemes [54, 81, 151].

Sophos

Sophos, proposed by Bost [52], is the first scheme putting forward privacy on top of the discussion, and whose performances degrade gracefully on scaling with the number of document, updates and search queries.

In Sophos, the client stores a state for each keyword. This state changes upon updates, when the same keyword is paired with a document on the server. The new state is not linkable to the previous update and search queries, and it is not revealed until a search query. Upon search, the state is provided to the server, who can correctly retrieve the matching documents. An old state would not produce the correct search result. To avoid the server to pre-compute future states, trapdoor permutations are used. Upon updates, no information is leaked to the server and the scheme is therefore *forward secure*.

For its importance on providing an acceptable balance between security and performance, Sophos became the motivating example of our mechanisation of searchable encryption schemes.

[1] This separation does not change the security scenario if both databases are outsourced to untrusted servers.

[2] This information was believed to do no harm in many dynamic schemes at the time, so it was acceptable to leak.

6.1.1 Formal methods and Searchable Encryption

Our security analysis, that we will illustrate in Section 6.4, is carried out in the computational model, introduced in Chapter 2. We mechanised the proof with the aid of EasyCrypt [26], a tool that employs formal methods to model cryptographic protocols, see Section 2.5.

As the computational model was required, other tools such as the Foundational Cryptographic Framework [138], CryptHOL [31], and CryptoVerif [45] have been considered. The definitions of Sophos are based on oracles, therefore they are *required* to model security properties against adaptive adversaries [137]. We introduced security against adaptive adversary in Section 2.2.5. This requirement drove our choice of which tool to use.

To the best of our knowledge, the only application of formal methods to Searchable Encryption is due to Petcher and Morrisett in 2015 [137], who provided a mechanised formal proof for the construction in [59] written in the Foundational Cryptographic Framework (FCF) [138]. They, however, support only constructions in functional style and security against non-adaptive adversary that prepares all the queries in advance. Almost nothing of their work can be reused to carry out proofs against adaptive adversaries; this limitation is because the proofs would be entirely different [137]. On the contrary, EasyCrypt allows for reasoning about imperative code, which is much more natural, as the most of the constructions in Searchable Encryption are written in imperative languages or pseudo-language: knowing a programming language would suffice to appreciate whether the model in EasyCrypt reflects the original construction.

Recently, Stoughton and Varia have shown the capability of EasyCrypt to carry out proofs in the Random Oracle Model against adaptive adversaries [154]. Similarly to their work, we explicitly limit the number of times that the adversary can call the oracle, allowing us, where possible, to get concrete upper bounds in the proofs without explicitly employing complexity theory and cryptographic assumptions. Differently from their work, we used the standard definitions of simulation-based paradigm [116], and we needed to refer to some cryptographic assumptions; in particular, the one-wayness of trapdoor permutations, which will be briefly introduced in Section 6.2.2. Our proof is valid for honest-but-curious adversary[3], and, by appropriately changing the client construction, the theorem can be turned to be capture malicious adversary [52]. In particular, the minimum leakage provided to the simulator is enough to produce computationally equivalent output as the original protocol. Therefore, the proof we produce is valid in absence of side channels and other sources of leakage.

CryptHOL supports reasoning in the ROM [118]. However, at the time we started and chosen what tool to work with, CryptHOL did not support simulation based proofs as opposed to EasyCrypt. Additionally, CryptHOL does not support imperative code reasoning, which we find much more natural to describe complex protocols, e.g. searchable encryption protocols, to check against the typical real implementations.

[3] A distrusted party of the protocol that at least plays the protocol as it is supposed to, but still tries to infer any information from running it.

CryptoVerif is a highly automated tool [45] where the protocols may be written in the pi calculus [5]. Even if its automation is very appealing, it applies to some proofs only; complex proofs deviating from the supported ones requires either interactive mode, very difficult to use, or changing the internal core of CryptoVerif. However it has been successfully used to prove security for popular TLS [48].

Some (on-paper) effort to support indistinguishability in the ROM, see Section 2.1.3, have been done in the symbolic model too [78]. However, the ideas they proposed have not been implemented in proof assistant tools. And their possible application to adaptive security in SE or the security guarantees one can prove for SE schemes are not clear and less intuitive to apply.

6.1.2 Notation and Definitions of Dynamic Searchable Encryption

Dynamic searchable encryption schemes have the practical functionality of letting the client update (add or delete) the documents. We notice that, naively, updates to documents can be always *added* to any searchable encryption scheme by simply re-starting everything with the freshly updated database. We consider dynamic schemes only those that do not need an almost full reinitialisation of the protocols, e.g. re-indexing all the documents or recreating secrets. For example, Curtmola et al. [71] define the scheme with five algorithms including the encryption of the database and focusing on searches only. Even if they discuss the updates (outside of their definitions), it cannot be considered a dynamic scheme, as it is required to re-initialise (or re-start), and the client needs to reconstruct all the secrets for newly inserted documents, following the idea of Chang et al. [60].

Several formal definitions have been given in the literature for different constructions [133, 52, 114, 151, 152], especially where a formal proof is provided or at least sketched. Even though they are fairly good for reasoning with the ad-hoc constructions in the related papers, none is obviously applicable to describe the schemes studied in other papers. Among the many definitions, we selected that of Naveed et al. [133] and Stefanov et al. [152], as we found them more easily comparable and clear to the point.

We stress that this comparison is not done to show that it is always *impossible* to adapt those definitions one another, but to show that those adaptations would generally require the security analysis (theorems and proofs) to be adapted, reconsidered or patched accordingly. We propose to solve this problem by using common definitions that provide primitives that can be used to model existing searchable schemes. Their impact can be especially appreciated when analysing different schemes or when reusing the same lemma or proof argument for a novel scheme.

Naveed et al. [133] define a dynamic scheme through five probabilistic polynomial-time procedures run by the client, some of which interact with the server (protocols), and two functionalities offered by the server, for uploading and downloading new documents. The definitions by Stefanov et al. [152] share similarities with the definitions in Bost [52], Lai et al. [114], or Song et al. [151], among others. Minor formal differences can be appreciated among them too: while Bost and Lai et al. describe

the scheme with an algorithm (for initial setup) and two protocols (for update and search), differently Stefanov et al. and Song et al. use three protocols; also, return values are different and arguments do not follow the same order.

Another aspect to consider is whether the algorithms make use of explicit state (Naveed et al.) or not (Song et al.): if the state is formally passed as argument the algorithm is stateless, as it does not need to have memory of the state[4].

To quickly appreciate those differences, Figure 6.2 reports an extract of the formal definitions of Naveed et al. and Stefanov et al.

Fig. 6.2 Comparison of definition for dynamic SSE schemes. The leftmost shows five stateful procedures, while the rightmost counts three stateless procedures.

Extract from Naveed et al. [133]

- SSE.keygen: Takes the security parameter as input, and outputs a key K_{SSE}. All of the following procedures take K_{SSE} as an input.
- SSE.indexgen: Takes as input the collection of all the documents (labeled using document IDs), a dictionary of all the keywords, and for each keyword, an *index file* listing the document IDs in which that keyword is present.[12] It interacts with the server to create a representation of this data on the server side.[13]
- SSE.search: Takes as input a keyword w, interacts with the server, and returns all the documents containing w.
- SSE.add: Takes as input a new document (labeled by a document ID that is currently not in the document collection), interacts with the server, and incorporates it into the document collection.
- SSE.remove: Takes as input a document ID, interacts with the server, and if a document with that ID is present in the server, removes it from the document collection.

[12] The index files can be created by SSE.indexgen, if it is not given as input.

[13] Typically, this would consist of a collection of (encrypted) documents, labeled by document indices (different from document IDs), and a representation of the index, which in our constructions will be stored using a blind-storage system.

Extract from Stefanov et al. [152]

Definition 1 (DSSE scheme)**.** *A dynamic searchable symmetric encryption (DSSE) scheme is a suite of three protocols that are executed between a client and a server with the following specification:*

- $(\mathsf{st}, D) \leftarrow \mathsf{Setup}((1^\lambda, N), (1^\lambda, \perp))$. *On input the security parameter λ and the number of document-keyword pairs N, it outputs a secret state* st *(to be stored by the client) and a data structure D (to be stored by the server);*
- $((\mathsf{st}', \mathcal{I}), \perp) \leftarrow \mathsf{Search}((\mathsf{st}, w), D)$. *The client input's include its secret state* st *and a keyword w; and server's input is its data structure D. At the end of the* Search *protocol, the client outputs a possibly updated state* st' *and the set of document identifiers $\mathcal{I}$ that contain the keyword w. The server outputs nothing.*
- $(\mathsf{st}', D') \leftarrow \mathsf{Update}((\mathsf{st}, \mathsf{upd}), D)$. *The client has input* st*, and an update operation* $\mathsf{upd} := (\mathsf{add}, \mathsf{id}, \mathbf{w})$ *or* $\mathsf{upd} := (\mathsf{del}, \mathsf{id}, \mathbf{w})$ *where* id *is the document identifier to be added or removed, and* $\mathbf{w} := (w_1, w_2, \ldots, w_k)$ *is the list of unique keywords in the document. The server's input is its data structure D. The* Update *protocol adds (or deletes) the document to (or from) D, and results in an updated client secret state* st' *and an updated server data structure D'.*

We now show how describing one scheme through the definitions of the other is far from straightforward. The first step is to use a single notation. We denote

- as λ the security parameter,
- as K the key generated by the initialisation procedures,
- as w a keyword from the list of all keywords W,
- as $\mathbf{DB}$ and $\mathbf{EDB}$ the plaintext database and its encryption (including constructions enabling for searches) respectively,
- as d a document index or document ID,
- as σ a secret state stored by the client,

[4] As separate objects, the server and the client are required to have their own state (for example, private keys or the encrypted database) regardless of their formalisation for reasoning with stateful or stateless procedures.

- as u a triplet for the update operation that can be $(\mathsf{add}, d, \mathbf{w})$ or $(\mathsf{del}, d, \mathbf{w})$ where $\mathbf{w}$ is a unique list of n keywords, and
- as $\mathcal{I}$ a map from keywords to matching document IDs, where $\mathcal{I}(w) \subseteq \mathcal{I}$ are the documents containing the keyword w.

A skeleton of the procedures by Naveed et al. and by Stefanov et al. with unified notation are illustrated in Algorithms 1 and 2 respectively. We opted to convert the definitions by Naveed et al. to be stateless. In their stateful form, they do not completely show what is the intended output affecting the server-side database description, nor the client's state σ. The client's state σ is a storage required to run the procedures and can include the key generated with keygen. Those interpretable parts have been completed by reading through the specific construction they propose and its security analysis.

Alg. 1. Prototype of definitions in Naveed et al. [133]

Naveed:

$\mathsf{keygen}(1^\lambda) \to K$ *(client only)*

$\mathsf{indexgen}(K, \mathbf{DB}, W, \mathcal{I}; \perp) \to \sigma; \mathbf{EDB}$

$\mathsf{search}(\sigma, w; \mathbf{EDB}) \to \mathcal{I}(w); \mathbf{EDB}'$

$\mathsf{add}(\sigma, d; \mathbf{EDB}) \to \sigma'; \mathbf{EDB}'$

$\mathsf{remove}(\sigma, d; \mathbf{EDB}) \to \sigma'; \mathbf{EDB}'$

Alg. 2. Prototype of definitions in Stefanov et al. [152]

Stefanov:

$\mathsf{setup}(1^\lambda, \mathbf{DB}; 1^\lambda, \perp) \to \sigma; \mathbf{EDB}$

$\mathsf{search}(\sigma, w; \mathbf{EDB}) \to \sigma', \mathcal{I}(w); \perp$

$\mathsf{update}(\sigma, u; \mathbf{EDB}) \to \sigma'; \mathbf{EDB}'$

The last step is to describe one construction through the definition of the other. In Naveed et al. the initialisation phase and the update operations are each split in two procedures.

The most relevant incompatibility is that Naveed.search modifies the index data structure, but Stefanov.search is required not to (no output). This *small* detail may break (at least) the correctness of the construction, as successive calls would use the old state. One may simply think of modifying the Stefanov.search procedure to return $\mathbf{EDB}'$; however, the security analysis in Stefanov et al. relies on that being $\perp$ when they prove security in a simulation-based fashion. Therefore, this modification would require the proof to be revised under this light. This difficulty, would suggest to *prefer* the definitions of Naveed et al.

In an attempt to lift the definitions of Stefanov et al. to those of Naveed et al., then another important incompatibility raises. In particular, one should be able to instantiate the initialisation procedures Naveed.keygen and Naveed.indexgen through the initialisation procedure Stefanov.search. This could be attempted in the way illustrated in Algorithm 3.

Alg. 3. An attempt to lift the definitions of Stefanov et al. [152] to those of Naveed et al. [133].

Naveed:

keygen (1^λ)
1 $(\sigma, \mathbf{EDB}) \leftarrow$ Stefanov.setup $\left(1^\lambda, \bot; 1^\lambda, \bot\right)$
2 $k \leftarrow$ extract the key from σ
3 **return** k

indexgen $(k, \mathbf{DB}, W, \mathcal{I}; \bot)$
4 $(\sigma, \mathbf{EDB}) \leftarrow$ Stefanov.setup $\left(1^\lambda, \mathbf{DB}; 1^\lambda, \bot\right)$
5 **forall** $w \in \mathcal{I}$ **do**
6 **forall** $i \in \mathcal{I}(w)$ **do**
7 $(\sigma, \mathbf{EDB}) \leftarrow$ Stefanov.update $((\sigma, (\mathsf{add}, i, w)); \mathbf{EDB})$
8 **return** $\sigma; \mathbf{EDB}$

In the initialisation part, Naveed.keygen and Naveed.indexgen must be called subsequently, as the key k is required for calling Naveed.indexgen. Then Stefanov.search will be run twice, re-creating the secrets (including k) so that the first may became obsolete. This surely breaks the construction of Stefanov et al. that needs to patched. One possible patch could be that of modifying the behaviour of the Stefanov.search, leveraging the fact that when it is called by Naveed.keygen, $\bot$ is passed in place of the database $\mathbf{DB}$ that could not be used. So, if no database is passed, the state σ is populate with the sole key, and $\mathbf{EDB}$ is not created; otherwise, Stefanov.search does not create a new key, but populates the rest of σ (if any) and $\mathbf{EDB}$. However, this would also require to modify the signature of Stefanov.search to allow for passing k (or σ) as a new argument. All the analysis done in Stefanov et al. must be reconsidered in the light of those modifications when described with the analysis of Naveed et al.

The above mentioned are examples of the difficulties in adopting existing definitions to describe different constructions of searchable encryption schemes. We discuss our solution to those difficulties in Section 6.3.1, where we provide general definitions in such a way that those constructions can be seen as instances or refinements of them. This will make not only easier to compare and reuse the analysis of different schemes, but also unify the description of the security claims of future constructions.

6.2 Cryptographic Primitives and Concepts

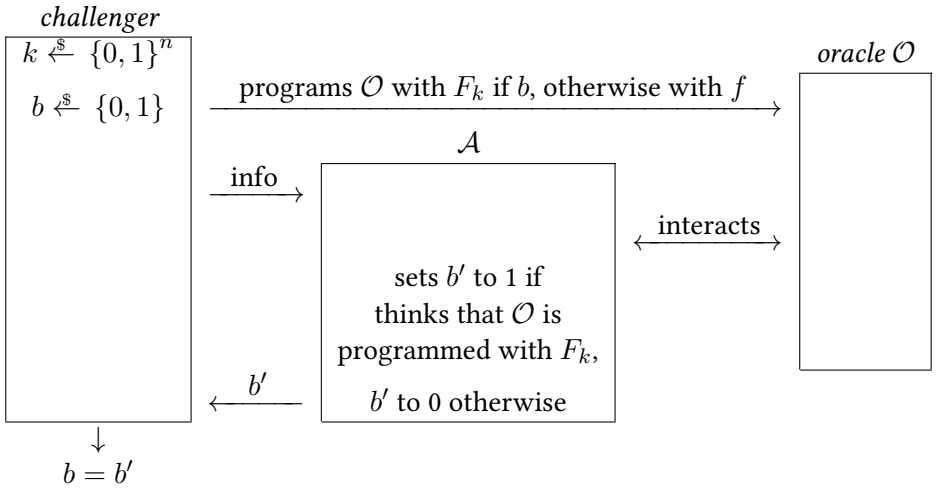
We provide a mechanised security analysis of $\Sigma o \varphi o \varsigma$, whose model is illustrated in Section 6.3.2, in terms of adaptive security to prove that the scheme is *forward secure*, as defined in Section 6.2.5. The cryptographic constructions required to model the protocol are Pseudo-Random Functions (PRF), hash functions, and one-way trapdoor permutations. Hash functions are modelled as random oracles, and formal definitions of Pseudo-Random Functions and trapdoor permutations are provided in Sections 6.2.1 and 6.2.2.

6.2.1 Pseudo-random functions

A family of pseudo-random functions is such that its functions, if randomly chosen, are indistinguishable from a random function, that is a function whose image is determined at random.

We borrow the definitions from Katz and Lindell [109], whose discussion refers to mapping strings of length n to strings of the same length for simplicity. We consider a random function $f : \{0,1\}^n \to \{0,1\}^n$ and a keyed function $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$, where the length of the strings n is the security parameter. We use $k \in \{0,1\}^\lambda$ to index the family F, whose elements are the functions $F_k : \{0,1\}^n \to \{0,1\}^n$. We define the security of F in terms of advantage. We employ a distinguishing experiment prf against an adversary $\mathcal{A}$ trying to distinguish between F_k and f, where preliminarily a key k for F is (uniformly) generated. We give the adversary the ability to interrogate an oracle $\mathcal{O}$ that can be programmed either with the truly random function f, denoted as $\mathcal{O}_f$, or the pseudo-random function F_k, denoted as $\mathcal{O}_{F_k}$. We write the security definition of indistinguishability according with the definition 2.2.4, where the experiment oExp is renamed as prf, as illustrated in Figure 6.3.

Fig. 6.3 Flow of the experiment for indistinguishability between F_k and f.



Definition 6.2.1 (Computationally secure pseudo-random function)**.** *The function F is a family of secure pseudo-random functions if and only if for any PPT adversary $\mathcal{A}$, there exists a negligible function μ such that*

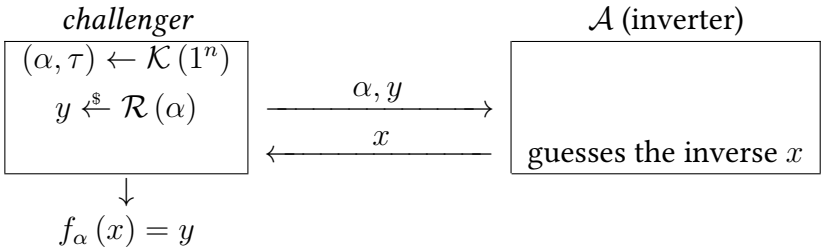
$$\mathbf{Adv}^{prf}_{F_k,f}\left(1^\lambda\right) = \left|\mathsf{Pr}\left[\mathcal{A}\left(\mathcal{O}_{F_k}\right) = 1\right] - \mathsf{Pr}\left[\mathcal{A}\left(\mathcal{O}_R\right) = 1\right]\right| \leq \mu\left(\lambda\right).$$

where k is chosen uniformly at random and n is the security parameter.

6.2.2 Collection of trapdoor permutations

A permutation is an invertible function where the domain and the codomain coincide, it basically *shuffles* the domain. A collection of permutations f can be obtained by

Fig. 6.4 Flow of the inverting experiment defining the security of trapdoor permutations.



indexing the domain D through the index α, D_α, that indexes the permutation f_α as well. The permutation f_α is called a *trapdoor* permutation if its values are easy to invert with the knowledge of a value τ (the trapdoor) associated to α. Otherwise if τ is not known, the permutation f_α cannot be efficiently inverted.

We follow the formalisation and, for the most, the notation of Lindell in [116]. We consider the sets A, T, and D_α, where A is the set for indexing the permutations f_α and its domain D_α, $\alpha \in A$, T is a set of trapdoors.

A collection of trapdoor permutations is defined by the quadruplet $(\mathcal{K}, \mathcal{R}, \mathcal{F}, \mathcal{B})$ of efficient algorithms, $\mathcal{K}$ for *key generation*, $\mathcal{R}$ for *random sampling*, $\mathcal{F}$ for *forward* mapping, and $\mathcal{B}$ for *backward* mapping. More formally,

- $\mathcal{K}(1^n)$ generates a pair (α, τ), of which τ is called the *trapdoor*;
- $\mathcal{R}(\alpha)$ samples both the domain and the codomain of f_α returning an uniformly distributed element;
- $\mathcal{F}_\alpha(x)$, where $x \in D_\alpha$, forward maps domain elements to codomain elements, $\mathcal{F}(\alpha, x) = f_\alpha(x)$;
- $\mathcal{B}_\tau(y)$, where $y \in D_\alpha$, backward maps codomain elements to domain elements, $\mathcal{B}(\tau, y) = f_\alpha^{-1}(y)$.

The security of a trapdoor permutation f_α is defined as the hardness of finding its inverse in the case the trapdoor τ is not unknown. In such context, the function f_α should be as secure as a one-way function; differently when τ is known, f_α must be efficiently invertible.

Formally, the security of f_α is defined through the inverting experiment iExp, illustrated in Figure 6.4, and described as follows:

- $\mathcal{K}(1^n)$ is run to obtain (α, τ), then $\mathcal{R}(\alpha)$ is run to choose a uniform y in the codomain D_α.
- The inverter $\mathcal{A}$ is given α and y as input, and output x.
- The output of the experiment is 1 if and only if $y = f_\alpha(x)$.

Definition 6.2.2 (Trapdoor permutation)**.** *A collection of* quasi *one-way permutations* $\pi = (\mathcal{K}, \mathcal{R}, \mathcal{F}, \mathcal{B})$ *is called a* trapdoor permutation *if the four algorithms* $\mathcal{K}$*,* $\mathcal{R}$*,* $\mathcal{F}$*,* $\mathcal{B}$

are efficient, i.e. they are probabilistic polynomial-time algorithms, and if for every PPT adversary $\mathcal{A}$, exists a negligible function μ such that:

$$\mathsf{Pr}\left[\mathsf{iExp}_{\pi}^{\mathcal{A}}\left(1^{n}\right)=1\right]\leq\mu\left(n\right).$$

where n is the security parameter. This probability corresponds to the following

$$\mathsf{Pr}\left[\mathcal{A}\left(1^{n},\alpha,y\right)=f_{\alpha}^{-1}\left(y\right)\right]$$

where y is a uniformly random value sampled with to $\mathcal{R}\left(\alpha\right)$, (α,τ) has been generated by calling $\mathcal{K}\left(1^{n}\right)$, and τ is unknown to $\mathcal{A}$.

The security Definition 6.2.2 is adapted from Katz and Lindell [109]. We remark that the minimum probability of inverter $\mathcal{A}$ is that of guessing from D_{α}, so $\left|D_{\alpha}\right|^{-1}$ has to be negligible too.

6.2.3 Database

The search functionality of searchable encryption scheme allows the client to retrieve a collection of documents that contain a selected keyword. The positions where the keyword lie in the document do not affect the search result. We work with index-based schemes, whose database offering the search functionality is detached from that of the actual documents. For the reasons above, the database of documents is usually simplified as a list of keyword-index pairs, or (equivalently) as a map from document indexes to set of keywords [59, 58, 133, 152, 52, 54, 151].

Definition 6.2.3 (Database)**.** *Given the set of all keywords $\{0,1\}^{\star}$ and a set of document indexes $\{0,1\}^{l}$, where l is an upper bound to the number of document indexes[5], then the (plaintext) database of documents $\mathbf{DB}$ is a map from document indexes to a subset of keywords.*

$$\mathbf{DB}:\{0,1\}^{l}\rightarrow W\subseteq\{0,1\}^{\star}.$$

The result of a search operation with some keyword $w\in\{0,1\}^{\star}$ is therefore $\{d|w\in\mathbf{DB}\left(d\right)\}$ and is sometimes shortly denoted as $\mathbf{DB}\left(w\right)$.

6.2.4 Leakage

The degree of confidentiality of a SSE scheme is determined by a the leakage function $\mathcal{L}=\left(\mathcal{L}^{\mathcal{I}},\mathcal{L}^{\mathcal{U}},\mathcal{L}^{\mathcal{S}}\right)$ which models all the information that is inherently revealed by running the protocols $\mathbf{Update}$ and $\mathbf{Search}$, $\mathcal{L}^{\mathcal{U}}$ and $\mathcal{L}^{\mathcal{S}}$ respectively, as well an initial leakage $\mathcal{L}^{\mathcal{I}}$ in the initialisation phase $\mathbf{Setup}$.

The leakage relates to file-access, search and update patterns that can be inferred by a (partial) history of incoming messages, that can be recorded by the server, while interacting with the client. Briefly, the file-access pattern leaks what encrypted files

[5]Some scheme may relate modified documents to different indexes, so that the number of deletion impact the number of index that can be used.

are downloaded from the database, search pattern relates search queries with the same keyword, and update pattern relate update operations upon the same keyword.

Definition 6.2.4 (History)**.** *We define the history* $\mathbf{Hist}$ *as the list of snapshots of the database* $\mathbf{DB}$*, paired with the update and search queries performed on it, indexed with the timestamp of such queries.*

$$\mathbf{Hist} = \{(i, \mathbf{DB}, e)\},$$

where i *is the timestamp,* $\mathbf{DB}$ *is the current database, and* e *is either an update query or a search query. An update query includes the operation and its operands, e.g.* $e = (\mathsf{add}, w, d)$ *with keyword* w *and index* d*, while a search query is the sole keyword* $e = w$*.*

We also define a function with the same name $\mathbf{Hist}\,(w)$ *as returning the subset of the history whose queries relate to the keyword* w*.*

$$\mathbf{Hist}\,(w) \stackrel{\text{def}}{=} \{(i, \mathbf{DB}, e) \in \mathbf{Hist} \wedge e \textit{ relates to } w\}.$$

The history is the internal state of the leakage function and must be *refined* to meet the confidentiality criteria desired.

To do that, following the definitions from [54], we derive from the history $\mathbf{Hist}$ both the *search pattern* and the *update pattern.* We do not use the file-access pattern in our discussion, as no interaction with the actual encrypted data storage is modelled.

Definition 6.2.5. *Given the history* $\mathbf{Hist}$*, we call the* search pattern $\mathbf{sp}\,(w)$*, related to a keyword* $w \in \{0,1\}^\star$*, the* ordered *collection of timestamps when a search was performed with* w*.*

$$\mathbf{sp}\,(w) \stackrel{\text{def}}{=} \{i |\, (i, \mathbf{DB}, w) \in \mathbf{Hist}\}.$$

Similarly, we call the update pattern $\mathbf{up}\,(w)$*, the* ordered *timestamps of update queries from the history that relate to* w*.*

$$\mathbf{up}\,(w) \stackrel{\text{def}}{=} \{i |\, (i, \mathbf{DB}, e) \in \mathbf{Hist}\,(w) \wedge e \textit{ is an update query}\}.$$

Finally, we call the query pattern $\mathbf{qp}\,(w)$*, the* ordered *timestamps of update queries from the history that relate to* w*.*

$$\mathbf{qp}\,(w) \stackrel{\text{def}}{=} \{i |\, (i, \mathbf{DB}, e) \in \mathbf{Hist}\,(w)\},$$

where d *is document index.*

We define the patterns to only include the timestamps, as the minimum leaked information from the patterns, as in Bost et al. [54] for the search and update pattern, and as in Bost [52] or Song et al. [151] for the query pattern. Depending on the amount of leakage of the schemes, their definition differ and is augmented with other information contained in the history; for example, the update pattern may also leak the operation, the document index, or other operands included in e from the history entry. In our discussion, the update pattern also leaks the document index, so we call it an *update-access pattern*, as it further leaks the access to file indexes for the update operations.

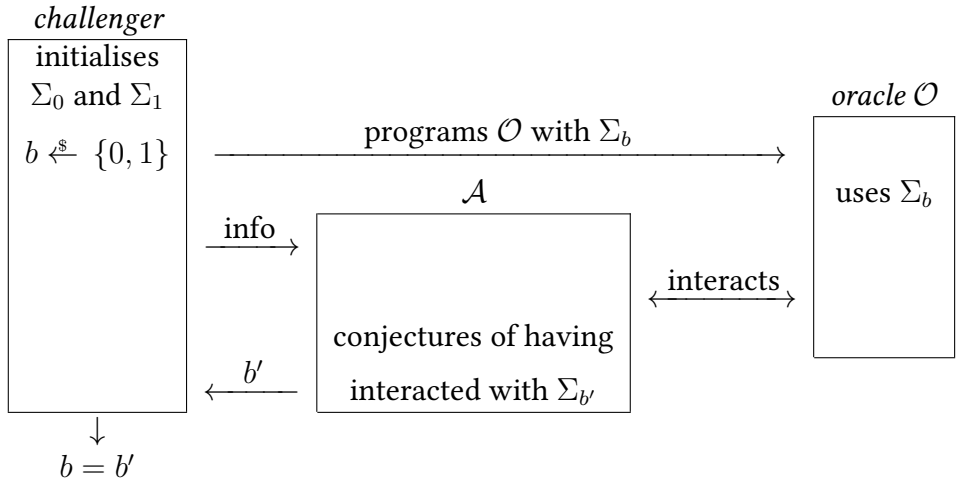
Definition 6.2.6. *Given the history* $\mathbf{Hist}$*, we call the* update-access pattern $\mathbf{uap}(w)$*, related to a keyword* $w \in \{0,1\}^\star$*, the* ordered *collection of pairs timestamp-index related to update operations.*

$$\mathbf{uap}(w) \stackrel{\text{def}}{=} \{(i,d) \,|\, (i,\mathbf{DB},e) \in \mathbf{Hist}(w) \wedge e \textit{ is an update query related to } d\}\,.$$

6.2.5 Adaptive security

The security of a searchable encryption scheme is based on the confidentiality of the client's data, notwithstanding the server is leaked some information. For dynamic schemes, security should be guaranteed against adaptive adversaries, as defined in Section 2.2.5. We define adaptive security for a SSE scheme Σ in terms of a particular distinguishing cryptographic experiment $\mathsf{SSE\,Exp}$ that follows the standard real-ideal experiment [86], specialising the Definition 2.2.4 of indistinguishability. An illustration of the experiment $\mathsf{SSE\,Exp}$ is shown in Figure 6.5. We denote as

Fig. 6.5 Adaptive security for the SSE scheme Σ with respect to the leakage function $\mathcal{L}$. The indistinguishability experiment $\mathsf{SSE\,Exp}$ is run between the games $\Sigma_0 = \Sigma$ and $\Sigma_1 = \Sigma_1(\mathsf{Sim},\mathcal{L})$, wrapping the simulator Sim and the leakage function $\mathcal{L}$.



$\mathsf{SSE\,Real}_{\Sigma}^{\mathcal{A},\mathcal{O}}$ the experiment, played against the distinguisher $\mathcal{A}$, in the case the oracle $\mathcal{O}$ is programmed with the real construction Σ; similarly, we denote as $\mathsf{SSE\,Ideal}_{\mathsf{Sim},\mathcal{L}}^{\mathcal{A},\mathcal{O}}$ the experiment in the case the oracle $\mathcal{O}$ is programmed with the ideal construction $\Sigma' = \Sigma'(\mathsf{Sim},\mathcal{L})$, that wraps together a simulator Sim and the leakage $\mathcal{L}$. In Σ', the leakage filters the original input values and produces the appropriate leakage to give as input to the simulator Sim. At the end, the simulator tries to produce an output that is computationally indistinguishable from the real construction. We follow the definition of [58] for adaptive security, and adapt it to the formality of simulation-based definitions and indistinguishability with oracles, as introduced in Section 2.2.3.

Definition 6.2.7 (Adaptive security)**.** *A dynamic SSE scheme* Σ *is* adaptively secure *with respect to a leakage function* $\mathcal{L}$*, if and only if for any PPT adversary* $\mathcal{A}$*, exist a PPT*

simulator Sim *and a negligible function* μ *such that*

$$\begin{aligned}\mathbf{Adv}^{\mathsf{SSE\,Exp}}_{\Sigma,\Sigma'(\mathsf{Sim},\mathcal{L})} &= \left|\mathsf{Pr}\left[\mathsf{SSE\,Real}_{\mathcal{A},\Sigma}\left(1^n\right)=1\right]-\mathsf{Pr}\left[\mathsf{SSE\,Ideal}_{\mathcal{A},\mathsf{Sim},\mathcal{L}}\left(1^n\right)=1\right]\right| \\ &\leq \mu\left(n\right)\end{aligned}$$

where n *is the security parameter, and* $\Sigma' = \Sigma'\left(\mathsf{Sim},\mathcal{L}\right)$ *is the construction built with the simulator and the leakage function.*

Forward security

Intuitively, forward security avoids updated keywords to be leaked when running an update query, so that the server cannot learn that the updated document matches a keyword previously queried. More formally, if a SSE scheme Σ is adaptively secure with respect to a leakage function $\mathcal{L}$ that leak at most the operation (add, delete, ...) when running the update protocol, then Σ is forward secure.

Definition 6.2.8 (Forward-security)**.** *A dynamic SSE scheme* Σ *is* forward secure*, if and only if* Σ *is adaptively secure with respect to a leakage function* $\mathcal{L}$ *where update calls do not leak more information than the operation* o *(e.g. add or delete) and the list of updated documents with the number of keywords updated in the documents (but not which ones).*

$$\mathcal{L}^{\mathcal{U}}\left(input\right) \subseteq \left\{o, \left\{\left(d_i, w_i\right)\right\}\right\}.$$

where $input$ *are all the parameters required to run the update protocol, and the set* $\left\{\left(d_i, w_i\right)\right\}$ *captures all updated documents as the number of keywords* w_i *modified in document* d_i*.*

We focus on forward security only; for definitions of the leakage function to guarantee backward security we refer to [54].

6.3 Modelling Searchable Encryption

6.3.1 Definitions of dynamic searchable encryption

Referring to the divergence of definition introduced in Section 6.1.2, the definitions we propose do not restrict schemes to follow index based schemes, where the server works with a database of documents' indexes rather than the (encrypted) documents themselves. From the point of view of the language, we relaxed algorithms' and protocols' arguments to allow for generic types, that can easily be refined depending on the specific construction to mechanise. Our code is therefore supportive for searchable encryption schemes in EasyCrypt, and we showed its effectiveness by instantiating Sophos with it and mechanising its forward security.

One way to describe a dynamic SSE scheme Σ is through the definition used for $\Sigma o\varphi o\varsigma$ in [52]. Other definitions have been used to support batch or file additions [112] and seldom is shown that single updates can be composed to achieve this functionality. However, the definition in [52] does not include those cases; therefore,

we define them in a more generic way. Dynamic SSE schemes are played by two entities, the server S and the client C, and are defined by the following.

- **Setup**, a protocol that initialises states and secrets of the client C and the server S.
- **Update**, a protocol played by C and S whose scope is to modify the (index) database by adding new pairs keyword-index or removing them.
- **Search**, a protocol played by C and S whose aim is to let S provide results with which C can reconstruct a list of all indexes whose documents contain a search keyword w.

The protocols may modify the secrets and the states of the client and the server; they can be implemented either by explicitly passing states (stateless) or implicitly (stateful).

We separate the algorithms run by the client from those run by the server, then we combine them to create the protocols.

Definition 6.3.1 (SSE Client)**.** *We define the client of a SSE scheme as the quadruplet* $C = (\mathcal{I}_c, \mathcal{U}_c, \mathcal{S}_c, \mathcal{O}_c)$ *such that:*

- $\mathcal{I}_c\left(1^\lambda, \mathbf{DB}\right)$ *initialises the local state* σ*. Optionally an output can be returned to be sent to the server, e.g. the encrypted database and initialisation values for the indexes.*
- $\mathcal{U}_c(o, in)$*, where* o *is the operation to perform, e.g. addition of a new keyword-index pair or deletion of an document, and* in *includes the operands of* o*. The output is sent to the server and include values to record the update securely on server-side.*
- $\mathcal{S}_c(w)$*, where* w *is the keyword to search. Its output is sent to the server and includes the tokens required for retrieving the search result.*
- $\mathcal{O}_c(x)$ *is an algorithm offering access to extra functionality. In particular, we use it to offer the two hash random oracles of the* $\Sigma o\varphi o\varsigma$ *implementation.*

Definition 6.3.2 (SSE Server)**.** *We define the server of a SSE scheme as the triplet* $S = (\mathcal{I}_s, \mathcal{U}_s, \mathcal{S}_s)$ *such that:*

- $\mathcal{I}_s(a)$ *where* a *is input received from the client. It creates a (possibly empty) database* $\mathbf{EDB}$*.*
- $\mathcal{U}_s(o, in)$ *where* o *is the operation to perform and* in *is the input sent by the client to perform such operation.*
- $\mathcal{S}_s(in)$ *where* in *is a message sent by the client required to carry out the search. It provides a return value from which the index list can be elaborated.*

Definition 6.3.3 (SSE Scheme)**.** *We define an SSE scheme as taking a client implementation* C *and a server implementation* S *as* $\Sigma(C, S) = (\mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{O})$*, where*

- $\mathcal{I}\left(1^{\lambda}\right)$ *models the initialisation algorithm* **Setup** *and should setup* C *and* S *by properly calling* $\mathcal{I}_c$ *and* $\mathcal{I}_s$*.*
- $\mathcal{U}\left(o, o_{in}\right)$ *models the protocol* **Update** *whose first algorithm is* $\mathcal{U}_c$ *and the last is* $\mathcal{U}_s$*, then it outputs the view of the protocol.*
- $\mathcal{S}\left(w\right)$ *models the protocol* **Search** *whose first algorithm is* $\mathcal{S}_c$ *and the last is* $\mathcal{S}_s$*, then it outputs the view of the protocol.*
- $\mathcal{O}\left(x\right)$ *is an algorithm offering access to extra functionality. It can simply reply the* $\mathcal{O}_c$ *from the client.*

Confidentiality of SSE schemes focus on the **Update** and **Search** phases. So, even if $\mathcal{I}_c$ and $\mathcal{I}_s$ may be modelled to play a communication protocol (as we do) like the other algorithms, its security is sometimes assumed as granted, hence the **Setup** phase simply runs smoothly and once. To show the usability of our definitions, we implemented the scheme $\Sigma o\varphi o\varsigma$ [52] as described in Section 6.3, then we proved its adaptive security as discussed in Section 6.5.

In many constructions, **Update** and **Search** are one-round protocols. For this reason what is returned from the client from $\mathcal{U}_c$ will be the exact input for $\mathcal{U}_s$, and similarly, the output of $\mathcal{S}_c$ will be exact input for $\mathcal{S}_s$. We show a generic model of one-round SSE protocols in Algorithm 4.

Alg. 4. Generic SSE protocol structure G where **Update** and **Search** are one-round protocols.

$C = (\mathcal{I}_c, \mathcal{U}_c, \mathcal{S}_c, \mathcal{O}_c)$, $S = (\mathcal{I}_s, \mathcal{U}_s, \mathcal{S}_s)$
$G\left(C, S\right) = (\mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{O})$

$\mathcal{I}\left(\lambda\right)$
  $r \leftarrow \mathcal{I}_c\left(\lambda\right)$
  $\mathcal{I}_s\left(r\right)$
  return r

$\mathcal{O} \leftarrow \mathcal{O}_c$

$\mathcal{U}\left(o, o_{in}\right)$
  $r \leftarrow \mathcal{U}_c\left(o, o_{in}\right)$
  if $r \neq \perp$ **then**
    $\mathcal{U}_s\left(o, r\right)$
  return r

$\mathcal{S}\left(w\right)$
  $w' \leftarrow \mathcal{S}_c\left(w\right)$
  if $w' \neq \perp$ **then**
    $r \leftarrow \mathcal{S}_s\left(w'\right)$
  else
    $r \leftarrow \emptyset$
  return r

Naveed and Stefanov modelled with our formalism

To correctly describe their models with our formalism, we need to *split* their descriptions into server-side and client-side operations. In fact, their formalism describes the whole protocol as in Definition 6.3.3. We denote

- as λ the security parameter,
- as K the key generated by the initialisation procedures,

- as w a keyword from the list of all keywords W,
- as $\mathbf{DB}$ and $\mathbf{EDB}$ the plaintext database and its encryption (including constructions enabling for searches) respectively,
- as d a document index or document ID,
- as $\mathbf{t}$ a list of tokens or tags, each relating both a document index and a keyword,
- as σ a secret state stored by the client,
- as u a triplet for the update operation that can be $(\mathsf{add}, d, \mathbf{w})$ or $(\mathsf{del}, d, \mathbf{w})$ where $\mathbf{w}$ is a unique list of keywords, and
- as $\mathcal{I}$ a map from keywords to matching document IDs, where $\mathcal{I}(w) \subseteq \mathcal{I}$ are the indexes of the documents containing the keyword w.

We illustrate the description of Naveed et al. [133] server-side and client-side in Algorithms 5, and analogous description of Stefanov et al. [152] in Algorithm 6 respectively.

Alg. 5. Prototype of definitions in Naveed et al. [133]

NC: // client	NS: // server
$\mathsf{keygen}(1^\lambda) \to K$	$\mathsf{setup}(\mathbf{EDB}) \to \mathbf{EDB}'$
$\mathsf{indexgen}(K, \mathbf{DB}, W, \mathcal{I}) \to \sigma; \mathbf{EDB}$	$\mathsf{search}(\mathbf{EDB}, \mathcal{I}(w)) \to \mathbf{EDB}'$
$\mathsf{search}(\sigma, w) \to \sigma, \mathcal{I}(w)$	$\mathsf{add}(\mathbf{EDB}, i, \mathbf{t}) \to \mathbf{EDB}'$
$\mathsf{add}(\sigma, d) \to \sigma, i, \mathbf{t}$	$\mathsf{remove}(\mathbf{EDB}, i) \to \mathbf{EDB}'$
$\mathsf{remove}(\sigma, d) \to \sigma$	

Alg. 6. Prototype of definitions in Naveed et al. [133]

SC: // client	SS: // server
$\mathsf{setup}(1^\lambda, N) \to \sigma$	$\mathsf{setup}(1^\lambda, N) \to \mathbf{EDB}$
$\mathsf{update}(\sigma, u) \to \sigma, \mathbf{t}$	$\mathsf{update}(\mathbf{EDB}, \mathbf{t}) \to \mathbf{EDB}'$
$\mathsf{search}(\sigma, w) \to \sigma, \mathbf{t}$	$\mathsf{search}(\mathbf{EDB}, \mathbf{t}) \to \mathbf{EDB}', \mathcal{I}(w)$

We show how our definitions can be instantiated with either the protocol proposed by Naveed et al. [133] or the one proposed by Stevanov et al. [152].

Alg. 7. Model of the protocol of Naveed et al. [133] in high level pseudo-code with our formalism.

Naveed:

Client state: σ

Server state: $\mathbf{EDB}$

Server:

proc $\mathcal{I}_s()$**:**
1 $\quad \mathbf{EDB} \leftarrow \mathsf{NS.setup}(\mathbf{EDB})$;

proc $\mathcal{U}_s(o, in)$**:**
2 $\quad$ **if** $o = \mathsf{add}$ **then**
3 $\quad\quad i, \mathbf{t} \leftarrow in$
4 $\quad\quad \mathbf{EDB} \leftarrow \mathsf{NS.add}(\mathbf{EDB}, i, \mathbf{t})$;
5 $\quad$ **else if** $o = \mathsf{del}$ **then**
6 $\quad\quad i \leftarrow in$
7 $\quad\quad \mathbf{EDB} \leftarrow \mathsf{NS.remove}(\mathbf{EDB}, i)$;

proc $\mathcal{S}_s(\mathcal{I}(w))$**:**
8 $\quad \mathbf{EDB} \leftarrow \mathsf{NS.search}(\mathbf{EDB}, \mathcal{I}(w))$;

Client:

proc $\mathcal{I}_c(1^\lambda)$**:**
1 $\quad k \leftarrow \mathsf{NC.keygen}(1^\lambda)$;
2 $\quad$ store k into σ;
3 $\quad W \leftarrow$ extract all keywords from $\mathbf{DB}$;
4 $\quad \sigma, \mathbf{EDB} \leftarrow \mathsf{NC.indexgen}(k, \mathbf{DB}, W, \perp)$;
5 $\quad$ **return** $\mathbf{EDB}$

proc $\mathcal{U}_c(o, d)$**:**
6 $\quad$ **if** $o = \mathsf{add}$ **then**
7 $\quad\quad \sigma, i, \mathbf{t} \leftarrow \mathsf{NC.add}(\sigma, d)$;
8 $\quad\quad$ **return** $(i, \mathbf{t})$
9 $\quad$ **else if** $o = \mathsf{del}$ **then**
10 $\quad\quad \sigma \leftarrow \mathsf{NC.remove}(\sigma, d)$;
11 $\quad\quad$ flag d into σ for later deletion;
12 $\quad\quad$ **return** $\perp$

proc $\mathcal{S}_c(w)$**:**
13 $\quad$ // NC.search() deletes documents
14 $\quad$ // in σ flagged for deletion too
15 $\quad \sigma, \mathcal{I}(w) \leftarrow \mathsf{NC.search}(\sigma, w)$;
16 $\quad$ **return** $\mathcal{I}(w)$

Alg. 8. Model of the protocol of Stefanov et al. [152] in high level pseudo-code with our formalism.

Stefanov:

Client state: σ

Server state: $\mathbf{EDB}$

Server:

proc $\mathcal{I}_s(1^\lambda, N)$**:**
1 $\quad \mathbf{EDB} \leftarrow \mathsf{SS.setup}(1^\lambda, N)$;

proc $\mathcal{U}_s(o, in)$**:**
2 $\quad \mathbf{t} \leftarrow in$
3 $\quad \mathbf{EDB} \leftarrow \mathsf{SS.update}(\mathbf{EDB}, \mathbf{t})$;

proc $\mathcal{S}_s(\mathbf{t})$**:**
4 $\quad \mathbf{EDB}, \mathcal{I}(w) \leftarrow \mathsf{SS.search}(\mathbf{EDB}, \mathbf{t})$;
5 $\quad$ **return** $\mathcal{I}(w)$

Client:

proc $\mathcal{I}_c(1^\lambda)$**:**
1 $\quad N \leftarrow$ extract number of keywords from $\mathbf{DB}$;
2 $\quad \sigma \leftarrow \mathsf{SC.setup}(1^\lambda, N)$;
3 $\quad$ **return** N

proc $\mathcal{U}_c(o, d, \mathbf{w})$**:**
4 $\quad \sigma, \mathbf{t} \leftarrow \mathsf{SC.update}(\sigma, o, d, \mathbf{w})$;
5 $\quad$ **return** $\mathbf{t}$

proc $\mathcal{S}_c(w)$**:**
6 $\quad \sigma, \mathbf{t} \leftarrow \mathsf{NC.search}(\sigma, w)$;
7 $\quad$ **return** $\mathbf{t}$

6.3.2 Sophos

As a case study, we formally model the protocol $\Sigma o\varphi o\varsigma$. The cryptographic constructions required to model the protocol are Pseudo-Random Functions (PRF), hash functions, and one-way trapdoor permutations. Hash functions are here modelled as random oracles. We modelled $\Sigma o\varphi o\varsigma$ as an instance of the definitions supporting SSE schemes discussed earlier in Section 6.3.1. While modelling, we needed to fix some slightly interpretable pseudo-code directives. For example, there was a directive "Output each ind" inside a loop in the **Search** protocol by the server side, which may be interpreted as new message sent to the client at every loop iteration, or alternatively as the **Search** protocol's output is somehow intermittent. To us this sounded a little odd; furthermore it is in clear contrast with the statement "both **Search** and **Update** are single round". Nevertheless we consider this detail of minor relevance, even if we needed to take them into account. Further similar minor details will not be discussed here, as of poor interest.

The model we implemented is illustrated in Algorithm 10. The construction we give (and therefore its model) is functionally equivalent to the original $\Sigma o\varphi o\varsigma$, that is reported in Algorithm 9.

Alg. 9. The original construction of $\Sigma o\varphi o\varsigma$, extract from [52].

$\underline{\mathsf{Setup}()}$

1: $K_S \xleftarrow{\$} \{0,1\}^\lambda$
2: $(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
3: $\mathbf{W}, \mathbf{T} \leftarrow$ empty map
4: **return** $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$

$\underline{\mathsf{Search}(w, \sigma; \mathsf{EDB})}$

Client:
1: $K_w \leftarrow F_{K_S}(w)$
2: $(ST_c, c) \leftarrow \mathbf{W}[w]$ ▷ $c = n_w - 1$
3: **if** $(ST_c, c) = \perp$
4: **return** $\emptyset$
5: Send (K_w, ST_c, c) to the server.
Server:
6: **for** $i = c$ **to** 0 **do**
7: $UT_i \leftarrow H_1(K_w, ST_i)$
8: $e \leftarrow \mathbf{T}[UT_i]$
9: $\mathsf{ind} \leftarrow e \oplus H_2(K_w, ST_i)$
10: Output each ind
11: $ST_{i-1} \leftarrow \pi_{\mathsf{PK}}(ST_i)$
12: **end for**

$\underline{\mathsf{Update}(\mathsf{add}, w, \mathsf{ind}, \sigma; \mathsf{EDB})}$

Client:
1: $K_w \leftarrow F(K_S, w)$
2: $(ST_c, c) \leftarrow \mathbf{W}[w]$
3: **if** $(ST_c, c) = \perp$ **then**
4: $ST_0 \xleftarrow{\$} \mathcal{M}$, $c \leftarrow -1$
5: **else**
6: $ST_{c+1} \leftarrow \pi^{-1}_{\mathsf{SK}}(ST_c)$
7: **end if**
8: $\mathbf{W}[w] \leftarrow (ST_{c+1}, c+1)$
9: $UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
10: $e \leftarrow \mathsf{ind} \oplus H_2(K_w, ST_{c+1})$
11: Send (UT_{c+1}, e) to the server.
Server:
12: $\mathbf{T}[UT_{c+1}] \leftarrow e$

We made the following two modifications on the client. First, we treat the stored value c in $W[w]$ differently, but with respect to its original *meaning*, that is the

Alg. 10. $\Sigma o\varphi o\varsigma$ model in pseudo-code; the highlighted parts points to the main differences with the original description.

$\Sigma o\varphi o\varsigma = G\left(C, S\right)$

Client local variables:
$k \in \{0,1\}^{\lambda}$
$\tau \in T$ // trapdoor
$W : \{0,1\}^{l} \to D_{\alpha} \times \mathbb{N}$

Server local variables:
$\alpha \in A$
$\mathbf{T} : \{0,1\}^{l} \to \{0,1\}^{m}$

Server:
proc $\mathcal{I}_s(\alpha)$:
1 $\quad \alpha \leftarrow \alpha$
2 $\quad \mathbf{T} \leftarrow \emptyset$

proc $\mathcal{U}_s(\mathsf{add}, t, e)$:
3 $\quad \mathbf{T}\left(t\right) \leftarrow e$

proc $\mathcal{S}_s(k_w, s, c)$:
4 $\quad r \leftarrow []$
5 $\quad i \leftarrow 0$
6 $\quad$ **while** $i < c$ **do**
7 $\quad\quad t \leftarrow H_1\left(k_w, s\right)$
8 $\quad\quad e \leftarrow H_2\left(k_w, s\right)$
9 $\quad\quad e \leftarrow e \oplus \mathbf{T}\left(t\right)$
10 $\quad\quad r \leftarrow e :: r$
11 $\quad\quad s \leftarrow \mathcal{F}_{\alpha}\left(s\right)$
12 $\quad\quad i \leftarrow i + 1$
13 $\quad$ **return** r

Client:
proc $\mathcal{I}_c(1^{\lambda})$:
1 $\quad k \xleftarrow{\$} \{0,1\}^{\lambda}$
2 $\quad \alpha, \tau \leftarrow \mathcal{K}\left(1^{\lambda}\right)$
3 $\quad W \leftarrow \emptyset$
4 $\quad$ **return** α

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
5 $\quad k_w \leftarrow F_k\left(w\right)$
6 $\quad$ **if** $w \notin W$ **then**
7 $\quad\quad s \xleftarrow{\$} D_{\alpha}$
8 $\quad\quad c \leftarrow 0$
9 $\quad$ **else**
10 $\quad\quad s, c \leftarrow W\left[w\right]$
11 $\quad\quad s \leftarrow \mathcal{B}_{\tau}\left(s\right)$
12 $\quad\quad c \leftarrow c + 1$
13 $\quad W\left[w\right] \leftarrow \left(s, c\right)$
14 $\quad t \leftarrow H_1\left(k_w, s\right)$
15 $\quad e \leftarrow i \oplus H_2\left(k_w, s\right)$
16 $\quad$ **return** $\left(t, e\right)$

proc $\mathcal{S}_c(w)$:
17 $\quad$ **if** $w \in W$ **then**
18 $\quad\quad k_w \leftarrow F_k\left(w\right)$
19 $\quad\quad s, c \leftarrow W\left[w\right]$
20 $\quad\quad r \leftarrow \left(k_w, s, c\right)$
21 $\quad$ **else**
22 $\quad\quad r \leftarrow \perp$
23 $\quad$ **return** r

proc $\mathcal{O}_c(i, k_w, s)$:
24 $\quad h \leftarrow \perp$
25 $\quad$ **if** $i = H_1$ **then**
26 $\quad\quad h \leftarrow \left(H_1\left(k_w, s\right), \perp\right)$
27 $\quad$ **else if** $i = H_2$ **then**
28 $\quad\quad h \leftarrow \left(\perp, H_2\left(k_w, s\right)\right)$
29 $\quad$ **return** h

number of documents relating to w that have been stored, subtracted by 1. We find more comfortable to use natural numbers[6], so lines 8 and 12 in the client side of Algorithm 10 have been properly modified, yet the c values stored in W are completely equivalent to the original ones. Second, we deferred the line 18 to be run only if required. Apart of being bad programming practise to do unnecessary computation, this line has unpleasant side effects when the simulation F is simulated by R. In particular, it stores a value into R's internal map and unnecessary overcomplicates the equivalence proof, as one side may sample a value in the **Update** protocol, while the other may sample the same value later in the **Search** protocol. This race situations can be solved using advanced proof tactics as the eager-lazy techniques, based on re-sampling and requiring extra constructions and difficult proofs [22].

We made the following two modifications on the server. First, we have converted the *for* loop to a *while* loop, see lines 6 and 12 in the server side of Algorithm 10. This change is simply because *for* loops are not supported in EasyCrypt, but luckily *while* loops are, and we could rewrite it equivalently. Second, in line 10 we explicitly collect all indexes before sending them to the client, or returning them as output of the **Search** protocol.

6.4 Formal Analysis of Forward Security

6.4.1 On the choice of a simulation-based proof in the presence of an adaptive adversary

Our formalisation supports definitions of security against adaptive adversary. An adversary is adaptive if the input to the queries provided by the scheme may be dependent on the result of previous queries, as opposed to a non-adaptive adversary that chooses all the input upfront.

In a first attempt, we modelled the *adaptive* behaviour with loops and traces. In particular, the adversary is given the current trace of execution and asked what operation to call next, so that her strategy can depend on the output from previous operations. This is done in a loop running at most a polynomial number of times. However, the support for loops in EasyCrypt was limited at the time, so we adopt a formalism where the adversary is provided with protocol functionalities as oracle calls[7]. The former construction find its theoretical basis on the real and ideal games in [86] where they show that this definition is equivalent to the latter with oracle accesses. Informally, the two are equivalent because the two operations (i) of asking the adversary what operation to perform and (ii) distinguishing the ensembles are abstract, so they could reflect any strategy of the adversary.

So in our formalism, adversaries are restricted to call external functions (as oracles) at most a polynomial number of times, and the distinguish operation of the adversary is abstract modelling for all adversaries. Being able to call the oracle during the

[6] They use integers starting from -1 and get mistaken in the algorithm for searches in the simulator.

[7] Our code provides both models, with loops and with oracles, but we focused on the latter only.

distinguish operation captures the *adaptive* adversary, as queries can be chosen upon any strategy, i.e. inputs to queries can depend on results of other queries.

The cryptographic experiment modelling adaptive security for searchable encryption schemes is illustrated in Algorithm 11. The structure of the experiment is that of

Alg. 11. Simulation-based experiment for adaptive security of a SSE scheme Σ. Here, Σ' is an SSE game wrapping the leakage function $\mathcal{L}$ and the simulator Sim. The oracle $\mathcal{O}$ can be programmed with either Σ or Σ' and provides black-box access to internal procedures $\Sigma.\mathcal{U}$, $\Sigma.\mathcal{S}$ and $\Sigma.\mathcal{O}$ or $\Sigma'.\mathcal{U}$, $\Sigma'.\mathcal{S}$ and $\Sigma'.\mathcal{O}$, respectively, but no access to the initialisation function is provided.

$\Sigma'\,(\mathsf{Sim}, \mathcal{L})$

proc $\mathcal{I}(\lambda)$**:**
1. $l \leftarrow \mathcal{L}.\mathcal{L}^{\mathcal{I}}(\lambda)$
2. $r \leftarrow \mathsf{Sim}.\mathcal{I}(l)$
3. **return** r

proc $\mathcal{U}(o, o_{in})$**:**
4. $l \leftarrow \mathcal{L}.\mathcal{L}^{\mathcal{U}}(o, o_{in})$
5. $r \leftarrow \mathsf{Sim}.\mathcal{I}(l)$
6. **return** r

proc $\mathcal{S}(w)$**:**
7. $l \leftarrow \mathcal{L}.\mathcal{L}^{\mathcal{S}}(w)$
8. $r \leftarrow \mathsf{Sim}.\mathcal{I}(l)$
9. **return** r

proc $\mathcal{O} = \mathsf{Sim}.\mathcal{O}$

$\mathsf{SSE\,Exp}\,(\Sigma, \mathsf{Sim}, \mathcal{L}, \mathcal{A})$

proc main**():**
1. $b \xleftarrow{\$} \{0,1\}$
2. $\Sigma' = \Sigma'\,(\mathsf{Sim}, \mathcal{L})$
3. **if** b **then**
4. $\quad \Sigma.\mathcal{I}(\lambda)$ `// real`
5. $\quad b' \leftarrow \mathcal{A}(\mathcal{O}_{\Sigma})$
6. **else**
7. $\quad \Sigma'.\mathcal{I}(\lambda)$ `// ideal`
8. $\quad b' \leftarrow \mathcal{A}(\mathcal{O}_{\Sigma'})$
9. **return** $b = b'$

an indistinguishability game, as defined in Definition 2.2.4. Here, since the leakage function is stateful, we found it intuitive to model it in algorithmic style, as it can enjoy an internal global state[8] and, during the proof, the internal state is concealed from the adversary.

Our contribution on the simulated protocol Σ' in Algorithm 11, along with the the experiment SSE Exp will make easier to instantiate security theorems for adaptive security of any index-based dynamic SSE scheme. As usual in the computational model, the most difficult part are the proofs. To show the effectiveness of our contribution, we instantiated the SSE protocol Sophos [52] and proved its forward security.

6.4.2 Forward security - Definition

Adapting its security definition in [52] to the Definition 6.2.8, the adaptive security of $\Sigma o \varphi o \varsigma$ is defined as:

[8]We could emulate stateful behaviour with immutable functional style: in that case, the function would have required to explicitly pass the state as an extra argument.

Theorem 6.4.1 (Forward security of $\Sigma o\varphi o\varsigma$)**.** *Given the SSE scheme $\Sigma o\varphi o\varsigma$ constructed with a collection of trapdoor permutations, a pseudo-random function, and two hash functions modelled as random oracles, and given the following leakage function*

$$\mathcal{L} = \left(\mathcal{L}^{\mathcal{I}}\left(1^{\lambda}\right), \mathcal{L}^{\mathcal{U}}\left(o, w, i\right), \mathcal{L}^{\mathcal{S}}\left(w\right)\right) = \left(\perp, \perp, \left(\mathbf{sp}\left(w\right), \mathbf{uap}\left(w\right)\right)\right)$$

where o is an operation, w is a keyword, i is a document index, $\perp$ stands for no leakage, $\mathbf{sp}$ *is the search pattern and* $\mathbf{uap}$ *is the update-access pattern as in Definitions 6.2.5 and 6.2.6. Then for all PPT adversary $\mathcal{A}$, exist a PPT simulator* Sim *and a negligible function μ such that*

$$\mathbf{Adv}^{\mathsf{SSE\,Exp}}_{\Sigma o\varphi o\varsigma,\mathsf{Sim},\mathcal{L}} \leq \mu\left(\lambda\right),$$

where λ is the security parameter.

One can argue about the leakage $\mathcal{L}^{\mathcal{U}}$ saying that the operation cannot be hidden to the server; however, such information is known even before engaging the $\mathbf{Update}$ protocol itself, being the addition the only supported operation. The server either does nothing if the operation is not an addition or, if malicious, whatever it does is out of the protocol and could potentially be done independently. Since the knowledge of the malicious server is not augmented by running the $\mathbf{Update}$ protocol, there is no leakage.

6.5 Forward security - Proof

In this section we show the proof extracts of *forward security* for the searchable encryption protocol $\Sigma o\varphi o\varsigma$ illustrated in Theorem 6.4.1 and whose construction is shown in Algorithm 10.

Proof structure

As the last part of our contribution, we show the most interesting and challenging extracts of the mechanised proof and differences with the original on-paper proof by Bost [52]. We do not discuss the full proof here as many parts would be redundant or exactly as the original proof. The structure of our discussion will touch the following aspects:

- Differences in the intermediate games between ours patched proof and the original proof with patches to overlooked gaps, Section 6.5.1.
- Game reduction strategy we adopted, Section 6.5.2.
- Sequence of computationally indistinguishable games that reaches the conclusion of the Theorem 6.4.1, Section 6.5.3.

The full description of intermediate games is provided at the end of Appendix B. To simplify some steps of the proof, we used the extension that we introduced in Chapter 5; however, we do not discuss them in detail because of they would introduce unnecessary complexity to the proof. Also, we developed that approach late in the

proof and changing everything in function of that would have been an engineering effort that would have not added much to the contribution itself.

6.5.1 Differences between the mechanised proof and the on-paper proof

While obviously we recognise and emphasise the goodness of the majority of the original proof, we also aim at highlighting what parts in the proofs we found to be source of imprecisions, and what gaps should not be taken for granted when carrying on simulation-based proofs. The main sources of imprecisions are three:

- random functions are treated as collision-free;
- permutations are treated as cycle-free; and
- indistinguishability is entailed by equivalence of a single procedure, rather than all procedures.

Also, it is not often that the effort of mechanising a proof in the computational model brings such aspects to the light. A counterexample that shows the incorrectness of an equivalence between two games is illustrated in Appendix B.1.

Since we have to prove Theorem 6.4.1, we needed to model the simulator and the leakage function (as an algorithm, as discussed in Section 6.3.1); the model of the simulator is illustrated in Algorithm 12, that can be compared to the original description in Algorithm 13. We omit the server's part, as it does not change across the whole proof.

Some changes have been done with respect to the original simulator in the proof of $\Sigma o\varphi o\varsigma$, as it was incorrect. First, we needed to adjust the correctness of early termination due to empty update pattern, see line 15 in Algorithm 12. In the original, when the update pattern had only one element, the simulator incorrectly returned with empty. Second, we only fill the table W when required, see line 21; if W is filled needlessly as in the original simulator, then the equivalence proof would be much more complex to formally carry out. Third, we fixed the timestamps updates in line 30. This change is very important to simulate the number of (supported) queries, from this number depends the correctness of the result; without it, would not be possible to produce probabilistically equivalent output as the real protocol. Finally, as we already explained for the model of $\Sigma o\varphi o\varsigma$ in Algorithm 10, we were required to transmute the *for* loop into an equivalent *while* loop, see line 19. Reaching the simulator game is only the last step of the proof, which counts 16 intermediate games.

Original proof highlights. The on-paper proof is structured as a sequence of intermediate games from the real execution to the ideal execution. At a certain point, the proof reaches a game simulating the hash functions and storing their value in two separate tables, one of which is lazily updated from the other in the **Search** protocol. The authors spotted two *bad events*, each of which appears in two places. As in many

Alg. 12. Model of the simulator Sim in pseudo-code - server parts are not shown as they equal previous games; the highlighted parts show the difference with the original description. The procedures f_k, f_w, SimH1 and SimH2 are random function implementations.

Simulator:

$(\alpha, \tau) \in A \times T$
$t \in \mathbb{N}$
$W : \mathbb{N} \to D_\alpha$
$F : \mathbb{N} \to K$
$T : \mathbb{N} \to \{0,1\}^m$
$E : \mathbb{N} \to \{0,1\}^l$
$T_{H_1} : K \times \{0,1\}^l \to \{0,1\}^m$
$T_{H_2} : K \times \{0,1\}^l \to \{0,1\}^l$

proc $f_k(w)$

proc $f_w(w)$

proc SimH1(k_w, s)

proc SimH2(k_w, s)

proc $\mathcal{I}(\perp)$:
1 $\quad \alpha, \tau \leftarrow \mathcal{K}(\lambda)$
2 $\quad t \leftarrow 0$ // timestamp
3 $\quad T \leftarrow W \leftarrow F \leftarrow E \leftarrow T_{H_1} \leftarrow T_{H_2} \leftarrow \emptyset$

proc $\mathcal{U}(\perp)$:
4 $\quad T(t) \in_R \{0,1\}^m$
5 $\quad E(t) \in_R \{0,1\}^l$
6 $\quad r \leftarrow (T(t), E(t))$
7 $\quad t \leftarrow t + 1$
8 $\quad$ **return** r

proc $\mathcal{O}(i, z)$:
9 $\quad h \leftarrow \perp$
10 $\quad$ **if** $i = H_1$ **then**
11 $\quad\quad h \leftarrow (\text{SimH1}(z), \perp)$
12 $\quad$ **else if** $i = H_2$ **then**
13 $\quad\quad h \leftarrow (\perp, \text{SimH2}(z))$
14 $\quad$ **return** h

proc $\mathcal{S}(\mathbf{sp}(w), \mathbf{uap}(w))$:
15 $\quad$ **if** $\mathbf{uap}(w) \neq \emptyset$ **then**
16 $\quad\quad \bar{w} \leftarrow \min\{\mathbf{sp}(w)\}$
17 $\quad\quad k_{\bar{w}} \leftarrow f_k(\bar{w})$
18 $\quad\quad i \leftarrow 0$
19 $\quad\quad$ **while** $i < |\mathbf{uap}(w)|$ **do**
20 $\quad\quad\quad$ **if** $i = 0$ **then**
21 $\quad\quad\quad\quad W(\bar{w}) \leftarrow f_w(\bar{w})$
22 $\quad\quad\quad\quad s \leftarrow W(\bar{w})$
23 $\quad\quad\quad$ **else**
24 $\quad\quad\quad\quad s \leftarrow \mathcal{B}_\tau(s)$
25 $\quad\quad\quad j, o, e \leftarrow \mathbf{uap}(w)[i]$
26 $\quad\quad\quad T_{H_1}(k_{\bar{w}}, s) \leftarrow T(j)$
27 $\quad\quad\quad T_{H_2}(k_{\bar{w}}, s) \leftarrow e \oplus E(j)$
28 $\quad\quad\quad i \leftarrow i + 1$
29 $\quad r \leftarrow$ as server would
30 $\quad t \leftarrow t + 1$
31 $\quad$ **return** r

Alg. 13. The original construction of the simulator in the proof of forward security of $\Sigma o\varphi o\varsigma$, extract from [52].

S.Setup()
1: $(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
2: $\mathbf{W}, \mathbf{T} \leftarrow$ empty map
3: $u \leftarrow 0$
4: **return** $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$

S.Update()
Client:
1: $\mathtt{UT}[u] \xleftarrow{\$} \{0,1\}^\mu$
2: $\mathsf{e}[u] \xleftarrow{\$} \{0,1\}^\lambda$
3: Send $(\mathtt{UT}[u], \mathsf{e}[u])$ to the server.
4: $u \leftarrow u + 1$

S.Search($\mathsf{sp}(w), \mathsf{Hist}(w)$)
Client:
1: $\overline{w} \leftarrow \min \mathsf{sp}(x)$
2: $K_{\overline{w}} \leftarrow \mathsf{Key}[\overline{w}]$
3: $ST_0 \leftarrow \mathbf{W}[\overline{w}]$
4: Parse $\mathsf{Hist}(w)$ as $[(u_0, \mathsf{add}, \mathsf{ind}_0), \ldots, (u_c, \mathsf{add}, \mathsf{ind}_c)]$
5: **if** $c = 0$
6: $\quad$ **return** $\emptyset$ $\quad$ 19
7: **for** $i = 0$ **to** c **do**
8: $\quad$ Program H_1 s.t. $H_1(K_{\overline{w}}, ST_i) \leftarrow \mathtt{UT}[u_i]$
9: $\quad$ Program H_2 s.t. $H_2(K_{\overline{w}}, ST_i) \leftarrow \mathsf{e}[u_i] \oplus \mathsf{ind}_i$
10: $\quad ST_{i+1} \leftarrow \pi^{-1}_{\mathsf{SK}}(ST_i)$
11: **end for**
12: Send $(K_{\overline{w}}, ST_c)$ to the server.

other proofs, the game copes with the bad events, to keep the consistency of the constructions. After that, they discharge the bad event reducing it to the hardness of inverting the trapdoor permutation. The last part reaches a game close enough to be shown indistinguishable from the simulator. Finally, the sequence of games is put together to have the final form of the theorem.

Our proof is very similar from the point of view of the sequence of games, but it differs at the point where we have to provide a game which copes with the bad events to keep consistency. The reason why we deviated is because we spotted six bad events and the complexity of the many tables read and written by different procedures called in an unpredictable order made too difficult to fix all the inconsistency issues coming from the *bad* events. We will explain how we circumvented the necessity of coping with consistency later in Section 6.5.2.

To make the reader's more comfortable while following our discussion on the most interesting parts of the proof, the Table 6.1 summarises the sequence of games from the construction of $\Sigma o\varphi o\varsigma$ to its simulation $\Sigma'(\mathsf{Sim}, \mathcal{L})$ that uses the simulator Sim and the leakage function $\mathcal{L}$.

Table 6.1 Summary of the sequence of games with the difference of construction with respect to the previous game in the chain.

$\Sigma o\varphi o\varsigma$	
$\sim G_1$	distinguish between PRF and a keyed RF
$\sim G_2$	distinguish between H1 and a Random Oracle
$\sim G_3$	distinguish between H2 and a Random Oracle
$\sim G_4$	distinguish between keyed RF and RF
$\leq G_5$	introduce bad events $b_c, b_t, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}$
$\leq G_6$	ignore b_{h_2}
$\leq G_7$	ignore b_{h_1}
$\leq G_8$	ignore b_t
$\leq G_9$	ignore b_{t_2}
$\leq G_{10}$	ignore b_{t_1}
$\leq G_{11}$	ignore b_c
$\sim G_{12}$	remove all bad events (no bad events assumed to happen)
$\sim G_{13}$	simulate result using update-access pattern
$\sim G_{14}$	remove unused RF (as its result is simulated)
$\sim G_{15}$	simulate result using search pattern
$\sim G_{16}$	remove an unused internal table (as simulated)
$\sim \Sigma'(\mathsf{Sim}, \mathcal{L})$	full simulation with Sim and the leakage function $\mathcal{L}$

Bad events. During the mechanisation of the proof, we could spot six bad events. We collect all bad events in the game G_5 which does *not* keep consistency, see Appendix B.2 for more details. We pay the price for this laziness by doubling the bad events handling, therefore the final probability which eventually stays negligible.

The first event we discuss, b_c, happens due to repetitions of the random function F replacing the PRF. The probability of each event happening is related to the birthday problem. The birthday problem studies the probability of having (at least)

two equal birthday dates in function of the number of people in the trial (assuming their independence). This is similar to that of finding collisions in a random function, and it is bounded by $q^2/|K|$ where q is the polynomial bound in the oracles, and K is the sampling set. This bad event is handled twice in games G_5 and G_{11}, so we have

$$\mathsf{Pr}\left[G_5\left(\mathcal{A}\right) : b_c\right] = \mathsf{Pr}\left[G_{11}\left(\mathcal{A}\right) : b_c\right] \leq \frac{q^2}{|K|}. \tag{6.1}$$

In the next parts of our discussion, the formulas will become more complex, so we introduce here a short notation we adopt from now on in the discussion. We use $\mathcal{A}_G$ for $\mathsf{Pr}\left[G\left(\mathcal{A}\right) = 1\right]$ and conveniently $\mathcal{A}_i$ for $\mathsf{Pr}\left[G_i\left(\mathcal{A}\right) = 1\right]$. Additionally, we denote the probability of the event e in the game G run against the adversary $\mathcal{A}$, $\mathsf{Pr}\left[G\left(\mathcal{A}\right) : e\right]$, as $\mathcal{A} : e$. So for example, $\mathcal{A}_i : G\left(\mathcal{A}\right)_i = 1$ is exactly $\mathcal{A}_i$ that is the probability of distinguishing result. So that the equation above can be re-written as

$$\mathcal{A}_5 : b_c = \mathcal{A}_{11} : b_c \leq \frac{q^2}{|K|}. \tag{6.2}$$

The second bad event, b_t, reflects the fact that trapdoor permutations may be affected by repetitions and cycles (up to fixed points). Clearly, if this event would not be negligible, the probability of being able to invert a values without knowing the trapdoor would become too high to be the trapdoor permutation considered secure. However, one-way functions have been studied for long to be used for pseudo-random generation [101], so we borrow the upper bound from the birthday problem again that applies to pseudo random generators. The bad event b_t may occur in our games G_5 and G_8, so we have

$$\mathcal{A}_5 : b_t = \mathcal{A}_8 : b_t \leq \frac{q^2}{|D_\alpha|} \tag{6.3}$$

The other bad events, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, are those related to the hardness to invert the trapdoor permutation originally spotted. The condition by which they set the bad event to true in the hash function simulation is incorrect. Interestingly enough, they silently fixed it in the more recent paper [54]. Even with this fix, the other bad events are not handled. Differently from their proof, we treat those events independently, the only risk is to overestimate the upper bound of the security of the scheme, but the most important goal is to show that the upper bound is a negligible function; in particular, they do not discuss one of the two occurrences of the bad event, so part of the proof was missing anyway. We do not re-propose here the proof of these bad events as it would be the same as in the original proof; hence, we limit to state the result. Finally, these bad events may happen in our games G_5, G_6, G_7, G_9, G_{10}.

$$\begin{aligned} &\mathcal{A}_5 : b_{h_1} = \mathcal{A}_5 : b_{h_2} = \mathcal{A}_5 : b_{t_1} = \mathcal{A}_5 : b_{t_2} = \mathcal{A}_6 : b_{h_2} = \mathcal{A}_7 : b_{h_1} = \\ &= \mathcal{A}_9 : b_{t_2} = \mathcal{A}_{10} : b_{t_1} \leq \mathsf{Pr}\left[\mathcal{A}\left(1^n, \alpha, r\right) = f_\alpha^{-1}\left(\mathcal{R}\left(\alpha, r\right)\right)\right]. \end{aligned} \tag{6.4}$$

We proved that all the bad events we shown here on paper are the exact distance between those games and either the previous or the next in the sequence of games. We will join all together in Section 6.5.3.

6.5.2 A different strategy in game reductions

Motivation

Reasoning about equivalence of probabilities related to output of algorithms is supported in EasyCrypt, and the proof of such theorems can be split with the rules briefly illustrated in Section 2.5. A major source of imprecision in the original proof was due the incorrect application of statistical equality of procedures, or their indistinguishability. In particular, they let suffice the output of a single oracle function to state the indistinguishability of two games. On the contrary, following the rule, in general one should prove the indistinguishability of the output of all the functions, plus other sub-goals, see the Rule 2.1 in Section 2.5.

The most challenging and time-consuming part of the proof is that of finding suitable invariant while proving equivalence (or equivalence up-to-bad-event) while comparing abstract procedures. We remark that usually on-paper proofs assume many obvious-looking steps, sometimes introducing big gaps to complete the proof faster. However, sometimes we forgive some cases and the gaps introduced become flaws in the proof. And a proof with errors is not a proof until they are fixed. The invariant can be very long and as such prone to be incomplete or incorrect: they are stated at the beginning of the goal to prove, and until we find an easier sub-goal to prove that highlight some error. This process takes its time.

The reason why we brought this argument here is because the invariant conditions in this proof are strictly related to the consistency. The great effort to find invariant allowed us to find, with the aid of EasyCrypt, all the inconsistency problems relatively quickly; patching them was more time consuming. In particular, we spend much time in the last game - the one with the simulator - and the game collecting all the bad events, G_5. To support the inherent difficulty of fixing all inconsistency problems with six bad events, we show in Table 6.2 the read and write accesses to internal maps and tables by all the procedures involved with *only two bad events and only simulating one hash function,* $H1$. With four additional bad events the table would have become even more complex. The complexity of Table 6.2 reflects the complexity of keeping the internal maps and tables consistent during the proofs[9], including all the cases when the read or write accesses are guarded by if conditions, lie inside loops, or are *hidden* through calls to other procedures.

In an indistinguishability proof, it is common to spot negligible events whose handling however may cause inconsistency in the data structures used that are storing, mirroring, or simulating other structures. It is intuitive to create an intermediate game which takes into account those (bad) events and, in the case they happen, fixes the consistency they may break. The intuition behind is that of stating an equality with the previous game, then comfortably discharge the bad events, i.e. reducing them to known theorems or cryptographic assumptions. If we consider the games G_1, $\widetilde{G_2}$, and G_3 and a negligible function μ, then formally we want to prove

$$\mathcal{A}_1 = \mathcal{A}_{\widetilde{G_2}} \ \wedge \ \mathcal{A}_{\widetilde{G_2}} \leq \mathcal{A}_3 + \mu \ \Rightarrow \ \mathcal{A}_1 - \mathcal{A}_3 \leq \mu,$$

[9] Those inconsistencies lead the distinguisher to tell the games apart.

Table 6.2 Table of accesses of maps in the original $\widetilde{G_2}$ (with our notation). H_1 is simulated by the client, who offers oracle access to it, i.e. the server can call it. Legend: "r" read access, "w" write access, *indirect access by calling H_1, †the operation is guarded (it may never run), the initialisation routines are in gray.

		F	W	$\mathbf{T}$	T_{H_1}	T
client	$\mathcal{I}_c$		w		w	w
	$\mathcal{U}_c$	r/w†	r/w		r/w*†	r*†/w
	$\mathcal{S}_c$	r/w†	r		w†	r†
server	$\mathcal{I}_s$			w		
	$\mathcal{U}_s$			w		
	$\mathcal{S}_s$	r*†	r*†	r	r*/w*†	r*†
	H_1	r†	r†		r/w†	r†

where $\widetilde{G_2}$ has all the bad events sorted up. Forcing the first equality may require the use of extra structures, or changes in the proof that then shall be reviewed in light of the new change. In the unfortunate case when we have to prove equality up to bad event with many procedures passed to an adversary as oracle access, we have to keep in mind the rule 2.1 in Section 2.5. In particular, this would require us to find *one* invariant that holds in *all* cases of procedures called.

Our strategy

Finding an invariant without coping with what happens in the bad events, hence keeping consistency, is expected to be easier, since we can introduce the invariants with the assumption that no bad events happen, therefore avoiding those cases. We show the algebraic steps that prove that our strategy can be used as an alternative strategy helping to circumvent the need for coping with consistency. Let's assume G_2 is a game with three bad events b_1, b_2, and b_3. Let us adopt the very short notation, G_i^j to denote a game identical to G_i apart from the removed bad event b_j, and similarly $\mathcal{A}_i^j$ denotes $\mathsf{Pr}\left[G_i^j\left(\mathcal{A}\right)=1\right]$.

This exercise becomes easier only under the assumption that alternatively showing that $\mathcal{A}_2^{1,2,3}-\mathcal{A}_3\leq\widetilde{\mu}$ is relatively simple, and is based on the fact that also easy is showing that two procedures differs only by the *if* of a bad event. We want to ultimately show that $\mathcal{A}_1-\mathcal{A}_3\leq\mu$, same as above. We build a G_2 with all the bad events guarded by *if* statements, but in those cases, we do nothing or return a very distinguishable output. The following should be relatively easy to prove, as in the bad events we can distinguish easily:

$$\mathcal{A}_1-\mathcal{A}_2\leq\mathcal{A}_2:\left(b_1\vee b_2\vee b_3\right),$$

where $\mathcal{A}_2:\left(b_1\vee b_2\vee b_3\right)$ is the event when at least one of the three bad events happens, but nothing is said about the probability of distinguishing.

Then we start removing the bad events one by one, as we would have done in the above case too, so this part does not really add anything.

$$\mathcal{A}_2 - \mathcal{A}_2^1 \leq \mathcal{A}_2 : b_1$$
$$\mathcal{A}_2^1 - \mathcal{A}_2^{1,2} \leq \mathcal{A}_2^1 : b_2$$
$$\mathcal{A}_2^{1,2} - \mathcal{A}_2^{1,2,3} \leq \mathcal{A}_2^{1,2} : b_3$$

Applying the union of events, we can rewrite:

$$\mathcal{A}_2 : (b_1 \vee b_2 \vee b_3) \leq \mathcal{A}_2 : b_1 + \mathcal{A}_2 : b_2 + \mathcal{A}_2 : b_3$$

where we removed the eventual intersections: we are not interested whether the events are independent or not, as long as all of them are eventually negligible. If we compose them up, we get:

$$\mathcal{A}_1 - \mathcal{A}_2^{1,2,3} \leq \sum_{i \in \{1,2,3\}} \mathcal{A}_2 : b_i + \mathcal{A}_2 : b_1 + \mathcal{A}_2^1 : b_2 + \mathcal{A}_2^{1,2} : b_3$$

Being based on the same bad events, we expect $\mathcal{A}_2 : b_2 = \mathcal{A}_2^1 : b_2$ and $\mathcal{A}_2 : b_3 = \mathcal{A}_2^{1,2} : b_3$.

$$\mathcal{A}_1 - \mathcal{A}_2^{1,2,3} \leq \sum_{i \in \{1,2,3\}} \mathcal{A}_2 : b_i + \mathcal{A}_2 : b_1 + \mathcal{A}_2^1 : b_2 + \mathcal{A}_2^{1,2} : b_3$$

Since the distance between $\mathcal{A}_2^{1,2,3}$ and $\mathcal{A}_3$ is easy to get and it is negligible, we finally have:

$$\mathcal{A}_1 - \mathcal{A}_3 \leq 2 \cdot \sum_{i \in \{1,2,3\}} \mathcal{A}_2 : b_i + \tilde{\mu} \leq \mu.$$

since the above sum of negligible functions is still negligible.

6.5.3 Finalising the proof

As a final result, we show the sequence of games that we proved, then we join them together and replace the bad event probabilities with those discussed above. A small detail in the mechanised proof is that we modelled $\Sigma o \varphi o \varsigma$ to accept generic hash functions, then we modelled them as random oracles. This adds two addends to the right hand of the final inequality, which can be removed since at last, they were assumed to be indistinguishable from random.

In summary (see Table 6.1), we started with the experiment simulating the PRF with a truly random function F, then, similarly, the experiment to distinguish between provided hash functions and random oracles. We built the game G_5 which does include all the bad events we spotted and since it does implement the alternative strategy discussed above, we continued by progressively removing the bad events. From the game G_{12} to the simulator, we have a sequence of equalities. If we call

$B = \{b_c, b_t, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}\}$ the set of all bad events, we can write:

$$\begin{aligned}
\mathcal{A}_{\Sigma o\varphi o\varsigma} - \mathcal{A}_1 &= \mathbf{Adv}^{prf}_{F,R}\left(1^\lambda\right) \\
\mathcal{A}_1 - \mathcal{A}_2 &= \mathbf{Adv}^{H_1}_{H_1,ROM_1}\left(1^\lambda\right) = 0 \\
\mathcal{A}_2 - \mathcal{A}_3 &= \mathbf{Adv}^{H_2}_{H_2,ROM_2}\left(1^\lambda\right) = 0 \\
\mathcal{A}_3 &= \mathcal{A}_4 \\
\mathcal{A}_4 - \mathcal{A}_5 &\leq \sum_{b\in B} \mathcal{A}_5 : b \\
\mathcal{A}_5 - \mathcal{A}_6 &\leq \mathcal{A}_6 : b_{h_2} \\
\mathcal{A}_6 - \mathcal{A}_7 &\leq \mathcal{A}_7 : b_{h_1} \\
\mathcal{A}_7 - \mathcal{A}_8 &\leq \mathcal{A}_8 : b_t \\
\mathcal{A}_8 - \mathcal{A}_9 &\leq \mathcal{A}_9 : b_{t_2} \\
\mathcal{A}_9 - \mathcal{A}_{10} &\leq \mathcal{A}_{10} : b_{t_1} \\
\mathcal{A}_{10} - \mathcal{A}_{11} &\leq \mathcal{A}_{11} : b_c \\
\mathcal{A}_{12} &= \mathcal{A}_{13} = \mathcal{A}_{14} = \mathcal{A}_{15} = \mathcal{A}_{16} = \mathcal{A}_{\Sigma'(\mathsf{Sim},\mathcal{L})}
\end{aligned}$$

where the advantage using random oracles for hash functions is 0 as we are in the ROM, so they are exactly implemented with random oracles. Putting all together, we have

$$\begin{aligned}
\mathcal{A}_{\Sigma o\varphi o\varsigma} - \mathcal{A}_{\Sigma'(\mathsf{Sim},\mathcal{L})} \leq\ & \mathbf{Adv}^{prf}_{F,R}\left(1^\lambda\right) + \\
& + k \cdot \mathsf{Pr}\left[\mathcal{A}\left(1^n, \alpha, r\right) = f_\alpha^{-1}\left(\mathcal{R}\left(\alpha, r\right)\right)\right] + \\
& + k' \frac{q^2}{|K|} + k'' \frac{q^2}{|D_\alpha|}
\end{aligned}$$

where k, k' and k'' are positive constants, so they are not jeopardising the negligibility of the second hand of the inequality.

The probability of distinguishing can therefore be done as small as required, i.e. it is negligible. Hence,

$$\Sigma o\varphi o\varsigma \overset{\mathrm{c}}{\equiv} \Sigma'\left(\mathsf{Sim}, \mathcal{L}\right).$$

This result does confirm that $\Sigma o\varphi o\varsigma$ is forward secure in accordance to the definition provided and our mechanisation patches the original proof, and we finally have

$$\mathbf{Adv}^{\mathsf{SSE\ Exp}}_{\Sigma o\varphi o\varsigma,\mathsf{Sim},\mathcal{L}} \leq \mu\left(\lambda\right).$$

□

6.6 Conclusion

We proposed new atomic definitions of algorithms to model searchable encryption schemes. We have shown that such definition can be combined to model existing scheme, in particular $\Sigma o\varphi o\varsigma$ [52].

With the aid of the tool EasyCrypt, we propose a proof of forward secrecy of the protocol $\Sigma o\varphi o\varsigma$ patching the original proof. We compare ours with the most relevant previous work by Petcher and Morrisett [137], where they analyse the security of a searchable encryption protocol. The proof they described in their paper was among the most complex mechanized cryptographic proofs that has been completed to date (2015): if we compare to their work, we expanded even further than them by adding random oracles and adaptive adversaries.

Chapter 7

Conclusion

In this dissertation, we investigated the potential of the state-of-the-art tools that allow to reason about the security of cryptographic protocol in either symbolic model and the computational model. Moreover, we contributed to push (some of) their limits a bit further to show novel mechanisation of old and recent protocols.

In the symbolic model, we conducted a formal analysis of the Simple Password Exponential Key Exchange (SPEKE) protocol, which is part of the standard ISO/IEC 11770-4. Our formalisation covers from its first specification [105] to our last (now included in the standard) which came along with its formal verification of many security properties [95, 104]. We used the formal tool ProVerif [49] do automatise the anaylsis; we pushed the tool to its limit by formalising a large number of security properties and attacks: correctness, secrecy of the pre-shared password, implicit and explicit key authentication, weak and strong entity authentication, perfect forward secrecy, bilateral unknown key-share, impersonation, session-swap and malleability.

In the computational model, we automated security proofs of commitment and sigma protocols. In particular we successfully implemented the Pedersen commitment scheme [136, 126] and the Schnorr protocol of zero knowledge proof of knowledge (as a sigma protocol). Furthermore, we implemented the proof of forward secrecy of the Sophos scheme [52], that is a very complex searchable encryption scheme. We automated the sequences of games from the real construction to a simulated construction, with a very complex proof that patches some flaws in the original proof. Moreover to simplify the proof, we extended the core logic of EasyCrypt with labelling of variables typical of information flow analysis. We relate to the existing strategy of lazy sampling [39], that has been mechanised in EasyCrypt, showing that our approach can simplify the proof in some cases; other proofs can benefit by our contribution.

7.1 Future work

When we modelled the SPEKE protocol, we analysed malleability by modelling it as correspondences. A limitation of our approach directly relates to the limitation of the symbolic model to cover the complete group theory, which is at the basis of the protocol. As such, we have been able to model malleability attacks based on double

exponentiation only. A future work towards improving the support of malleability attacks can be that of extending the equational theory of ProVerif to allow for more comprehensive state space of equalities.

Inspired by a previous model of SPEKE [160] that did not find any attacks to the protocol, another interesting extension to our work can be that of modelling the same protocol and its security properties in various other languages, e.g. Tamarin. This would be a very interesting task to compare verification results and attacks across different tools. This might be done by extending existing tools that export to other languages [74, 13, 16].

We consider our work on commitment schemes comprehensive of all the desirable properties that can be automatically verified in the computational model; conversely, we limited our analysis of sigma protocols to the basic properties of stand-alone sigma protocols: completenss, special soundness and special honest verifier zero knowledge. We did not have the time to cover properties related to the composition of sigma protocols. As an ongoing work, we already implemented and proved some of those properties in EasyCrypt[1], and plan to generate trustable executable code from the EasyCrypt schemes for zero-knowledge protocols.

We mechanised the sequence of games in the proof of forward secrecy for the protocol Sophos [52], from the real construction to the ideal simulated construction. This allowed us to write the final theorem in terms of advantage of breaking hardness assumptions that relate to failure events. One bit of the mechanisation that we reserve as future work are the reductions of (some of) the single failure events to cryptographic assumptions, i.e. hardness of inverting trapdoor permutations.

Another important extension to our work relates to the information flow techniques that we implemented at the core of EasyCrypt, that analyses all-or-none leakage. An obvious improvement would be an extension that cope with *partial leakage*. Even though our implementation can easily be extended with definitions for partial leakage, we expect that already existing tactics would need to be refined and new tactics would need to be implemented. Another important extension would be to support more fundamental types and data structures. We only support basic types and maps, so an interesting future work can be the extension of our tactics to support *lists*, *tuples* and other data structures that currently are not supported.

[1] In collaboration with Benjamin Gregoire at INRIA.

References

[1] (2003). *Entrust TruePass™Product Portfolio Strong Authentication, Digital Signatures and end-to-end encryption for the Web Portal.* Entrust.

[2] (2016). *Security Note, BlackBerry 10 Devices.* BlackBerry Unlimited.

[3] (retrieved Jan 2017). *EasyCrypt Reference Manual.*

[4] Abadi, M. and Blanchet, B. (2005). Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, 58(1-2):3–27.

[5] Abadi, M., Blanchet, B., and Fournet, C. (2016). The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *arXiv preprint arXiv:1609.03003*.

[6] Abadi, M. and Rogaway, P. (2002). Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of cryptology*, 15(2):103–127.

[7] Abadi, M. and Tuttle, M. R. (1991). A semantics for a logic of authentication. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 201–216. ACM.

[8] Abdalla, M., Benhamouda, F., and MacKenzie, P. (2015). Security of the J-PAKE password-authenticated key exchange protocol. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 571–587. IEEE.

[9] Abdalla, M. and Pointcheval, D. (2005). Simple Password-Based Encrypted Key Exchange Protocols. In *CT-RSA*, volume 3376, pages 191–208. Springer.

[10] Almeida, J. B., Baritel-Ruet, C., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Oliveira, T., Stoughton, A., and Strub, P.-Y. (2019). Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1622.

[11] Ambrona, M., Barthe, G., and Schmidt, B. (2016). Automated Unbounded Analysis of Cryptographic Constructions in the Generic Group Model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 822–851. Springer.

[12] Armando, A., Arsac, W., Avanesov, T., Barletta, M., Calvi, A., Cappai, A., Carbone, R., Chevalier, Y., Compagna, L., Cuéllar, J., et al. (2012). The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–282. Springer.

[13] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P. H., Héam, P.-C., Kouchnarenko, O., Mantovani, J., et al. (2005). The AVISPA tool for the automated validation of internet security protocols and applications. In *International Conference on Computer Aided Verification*, pages 281–285. Springer.

[14] Armando, A. and Compagna, L. (2004). SATMC: a SAT-based model checker for security protocols. In *European Workshop on Logics in Artificial Intelligence*, pages 730–733. Springer.

[15] Armando, A., Compagna, L., and Ganty, P. (2003). SAT-based model-checking of security protocols using planning graph analysis. In *International Symposium of Formal Methods Europe*, pages 875–893. Springer.

[16] Arnaboldi, L. and Metere, R. (2019). Poster: Towards a Data Centric Approach for the Design and Verification of Cryptographic Protocols. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2585–2587.

[17] Backes, M., Pfitzmann, B., and Waidner, M. (2003). A composable cryptographic library with nested operations. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 220–230. ACM.

[18] Bana, G. and Comon-Lundh, H. (2014). A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 609–620. ACM.

[19] Barker, E., Chen, L., Roginsky, A., and Smid, M. (2013). Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography. *NIST special publication*, 800:56A.

[20] Barthe, G., Danezis, G., Grégoire, B., Kunz, C., and Zanella-Beguelin, S. (2013). Verified computational differential privacy with applications to smart metering. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 287–301. IEEE.

[21] Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., and Strub, P.-Y. (2014a). Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, pages 146–166. Springer.

[22] Barthe, G., Dupressoir, F., Grégoire, B., Schmidt, B., and Strub, P.-Y. (2014b). Computer-aided cryptography: some tools and applications. *Proc. All about Proofs, Proofs for All*.

[23] Barthe, G., Eilers, R., Georgiou, P., Gleiss, B., Kovács, L., and Maffei, M. (2019). Verifying relational properties using trace logic. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 170–178. IEEE.

[24] Barthe, G., Fournet, C., Grégoire, B., Strub, P.-Y., Swamy, N., and Zanella-Béguelin, S. (2014c). Probabilistic relational verification for cryptographic implementations. In *ACM SIGPLAN Notices*, volume 49, pages 193–205. ACM.

[25] Barthe, G., Grégoire, B., and Béguelin, S. Z. (2012). Probabilistic relational Hoare logics for computer-aided security proofs. In *International Conference on Mathematics of Program Construction*, pages 1–6. Springer.

[26] Barthe, G., Grégoire, B., Heraud, S., and Béguelin, S. Z. (2011). Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*, pages 71–90. Springer.

[27] Barthe, G., Grégoire, B., and Zanella Béguelin, S. (2009). Formal certification of code-based cryptographic proofs. *ACM SIGPLAN Notices*, 44(1):90–101.

[28] Barthe, G., Hedin, D., Béguelin, S. Z., Grégoire, B., and Heraud, S. (2010). A machine-checked formalization of Sigma-protocols. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 246–260. IEEE.

[29] Basin, D., Mödersheim, S., and Vigano, L. (2003). An on-the-fly model-checker for security protocol analysis. In *European Symposium on Research in Computer Security*, pages 253–270. Springer.

[30] Basin, D., Mödersheim, S., and Vigano, L. (2005). OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208.

[31] Basin, D. A., Lochbihler, A., and Sefidgar, S. R. (2017). CryptHOL: Game-based Proofs in Higher-order Logic. *IACR Cryptology ePrint Archive*, 2017:753.

[32] Basin, D. A., Lochbihler, A., and Sefidgar, S. R. (2020). CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, pages 1–73.

[33] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. (1997). A concrete security treatment of symmetric encryption. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE.

[34] Bellare, M., Desai, A., Pointcheval, D., and Rogaway, P. (1998). Relations among notions of security for public-key encryption schemes. In *Annual International Cryptology Conference*, pages 26–45. Springer.

[35] Bellare, M., Pointcheval, D., and Rogaway, P. (2000). Authenticated key exchange secure against dictionary attacks. In *international conference on the theory and applications of cryptographic techniques*, pages 139–155. Springer.

[36] Bellare, M. and Rogaway, P. (1993a). Entity authentication and key distribution. In *Annual international cryptology conference*, pages 232–249. Springer.

[37] Bellare, M. and Rogaway, P. (1993b). Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73.

[38] Bellare, M. and Rogaway, P. (2004a). Code-Based Game-Playing Proofs and the Security of Triple Encryption. *IACR Cryptology ePrint Archive*, 2004(331).

[39] Bellare, M. and Rogaway, P. (2004b). The game-playing technique. *International Association for Cryptographic Research (IACR) ePrint Archive: Report*, 331:2004.

[40] Bellare, M. and Rogaway, P. (2006). The security of triple encryption and a framework for code-based game-playing proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 409–426. Springer.

[41] Bellovin, S. M. and Merritt, M. (1992). Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*, pages 72–84. IEEE.

[42] Bender, J., Dagdelen, Ö., Fischlin, M., and Kügler, D. (2012). The PACE| AA Protocol for Machine Readable Travel Documents, and Its Security. In *Financial Cryptography*, volume 7397, pages 344–358. Springer.

[43] Benton, N. (2004). Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM.

[44] Blanchet, B. (2004). Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 86–100. IEEE.

[45] Blanchet, B. (2008). A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207.

[46] Blanchet, B. (2009). Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434.

[47] Blanchet, B. (2012). Security protocol verification: Symbolic and computational models. In *Proceedings of the First international conference on Principles of Security and Trust*, pages 3–29. Springer-Verlag.

[48] Blanchet, B. (2018). Composition theorems for CryptoVerif and application to TLS 1.3. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 16–30. IEEE.

[49] Blanchet, B. et al. (2001). An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *csfw*, volume 1, pages 82–96.

[50] Blanchet, B. et al. (2016). Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135.

[51] Boichut, Y., Kosmatov, N., and Vigneron, L. (2006). Validation of Prouvé protocols using the automatic tool TA4SP. *TFIT*, 6:467–480.

[52] Bost, R. (2016). Sophos: Forward Secure Searchable Encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1143–1154, New York, NY, USA. ACM.

[53] Bost, R. (2018). *Algorithmes de recherche sur bases de données chiffrées*. PhD thesis, Rennes 1.

[54] Bost, R., Minaud, B., and Ohrimenko, O. (2017). Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1465–1482, New York, NY, USA. ACM.

[55] Butler, D., Aspinall, D., and Gascón, A. (2020). Formalising oblivious transfer in the semi-honest and malicious model in CryptHOL. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 229–243.

[56] Canetti, R., Stoughton, A., and Varia, M. (2019). EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *32nd IEEE Computer Security Foundations Symposium (CSF 2019)*.

[57] Carré, B. and Garnsworthy, J. (1990). SPARK - an annotated Ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA'90*, pages 392–402.

[58] Cash, D., Jaeger, J., Jarecki, S., Jutla, C. S., Krawczyk, H., Rosu, M.-C., and Steiner, M. (2014). Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, volume 14, pages 23–26. Citeseer.

[59] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., and Steiner, M. (2013). Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual Cryptology Conference*, pages 353–373. Springer.

[60] Chang, Y.-C. and Mitzenmacher, M. (2005). Privacy preserving keyword searches on remote encrypted data. In *International Conference on Applied Cryptography and Network Security*, pages 442–455. Springer.

[61] Chase, M. and Kamara, S. (2010). Structured encryption and controlled disclosure. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 577–594. Springer.

[62] Chen, L. and Tang, Q. (2008). Bilateral Unknown Key-Share Attacks in Key Agreement Protocols. *J. UCS*, 14(3):416–440.

[63] Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P. H., Mantovani, J., Mödersheim, S., and Vigneron, L. (2004). A high level protocol specification language for industrial security-sensitive protocols. In *HAL Inria Archive*.

[64] Chevalier, Y. and Vigneron, L. (2001). A tool for lazy verification of security protocols. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 373–376. IEEE.

[65] Comon-Lundh, H. and Cortier, V. (2003). Security properties: two agents are sufficient. In *European Symposium On Programming*, pages 99–113. Springer.

[66] Corin, R. and Etalle, S. (2002). An improved constraint-based system for the verification of security protocols. In *International Static Analysis Symposium*, pages 326–341. Springer.

[67] Cortier, V., Rusinowitch, M., and Zălinescu, E. (2006). Relating two standard notions of secrecy. In *International Workshop on Computer Science Logic*, pages 303–318. Springer.

[68] Cramer, R. (1996). Modular design of secure yet practical cryptographic protocol. *PhD thesis, University of Amsterdam*.

[69] Cremers, C., Horvat, M., Scott, S., and van der Merwe, T. (2016). Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 470–485. IEEE.

[70] Cremers, C. J. (2008). The Scyther Tool: Verification, falsification, and analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 414–418. Springer.

[71] Curtmola, R., Garay, J., Kamara, S., and Ostrovsky, R. (2011). Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934.

[72] Dawn Xiaoding Song, Wagner, D., and Perrig, A. (2000). Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, pages 44–55.

[73] Denker, G., Meseguer, J., and Talcott, C. (1998). Protocol specification and analysis in Maude. In *Proc. of Workshop on Formal Methods and Security Protocols*, volume 25. Indianapolis, Indiana.

[74] Denker, G. and Millen, J. (2000). CAPSL integrated protocol environment. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 1, pages 207–221. IEEE.

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

[76] Dolev, D. and Yao, A. (1983). On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208.

[77] Durgin, N., Lincoln, P., Mitchell, J., and Scedrov, A. (2004). Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311.

[78] Ene, C., Lakhnech, Y., et al. (2009). Formal indistinguishability extended to the random oracle model. In *European Symposium on Research in Computer Security*, pages 555–570. Springer.

[79] Escobar, S., Hendrix, J., Meadows, C., and Meseguer, J. (2007). Diffie-Hellman cryptographic reasoning in the Maude-NRL protocol analyzer. *Proc. of SecRet*, 2007.

[80] Escobar, S., Meadows, C., and Meseguer, J. (2006). A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1):162–202.

[81] Etemad, M., Küpçü, A., Papamanthou, C., and Evans, D. (2018). Efficient dynamic searchable encryption with forward privacy. *Proceedings on Privacy Enhancing Technologies*, 2018(1):5–20.

[82] Fortnow, L. (2009). The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86.

[83] Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178.

[84] Goh, E.-J. et al. (2003). Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216.

[85] Goldreich, O. (2007). *Foundations of cryptography: volume 1, basic tools*. Cambridge university press.

[86] Goldreich, O. (2009). *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.

[87] Goldreich, O. and Krawczyk, H. (1996). On the composition of zero-knowledge proof systems. *SIAM Journal on Computing*, 25(1):169–192.

[88] Goldreich, O. and Lindell, Y. (2001). Session-key generation using human passwords only. In *Annual International Cryptology Conference*, pages 408–432. Springer.

[89] Goldwasser, S. and Micali, S. (1984). Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299.

[90] Gordon, A. D. and Jeffrey, A. (2001). Types for cyphers: thwarting mischief and malice with type theory. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 136–136.

[91] Goubault-Larrecq, J. (2000). A method for automatic cryptographic protocol verification. In *International Parallel and Distributed Processing Symposium*, pages 977–984. Springer.

[92] Grimm, N., Maillard, K., Fournet, C., Hriţcu, C., Maffei, M., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N., and Zanella-Béguelin, S. (2018). A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 130–145.

[93] Halevi, S. (2005). A plausible approach to computer-aided cryptographic proofs. *IACR Cryptology ePrint Archive*, 2005:181.

[94] Hao, F. (2010). On Robust Key Agreement Based on Public Key Authentication. In *Financial Cryptography*, volume 6052, pages 383–390. Springer.

[95] Hao, F., Metere, R., Shahandashti, S. F., and Dong, C. (2018). Analyzing and patching SPEKE in ISO/IEC. *IEEE Transactions on Information Forensics and Security*, 13(11):2844–2855.

[96] Hao, F. and Ryan, P. Y. (2008). Password authenticated key exchange by juggling. In *International Workshop on Security Protocols*, pages 159–171. Springer.

[97] Hao, F. and Shahandashti, S. F. (2014). The SPEKE Protocol Revisited. *SSR*, 14:26–38.

[98] Harkins, D. (2008). Simultaneous authentication of equals: a secure, password-based key exchange for mesh networks. In *Second International Conference on Sensor Technologies and Applications, 2008 (SENSORCOMM'08)*, pages 839–844. IEEE.

[99] Hazay, C. and Lindell, Y. (2010). *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media.

[100] IEEE P1363.2:D26 (2006). Standard specifications for password-based public-key cryptographic techniques. Standard, Institute of Electrical and Electronics Engineers, Inc.

[101] Impagliazzo, R., Levin, L. A., and Luby, M. (1989). Pseudo-random generation from one-way functions. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 12–24.

[102] Islam, M. S., Kuzu, M., and Kantarcioglu, M. (2012). Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Ndss*, volume 20, page 12. Citeseer.

[103] ISO/IEC 11770-4:2006 (2006). Information technology - security techniques - key management - part 4: Mechanisms based on weak secrets. Standard, International Organization for Standardization, Geneva, CH.

[104] ISO/IEC 11770-4:2017 (2017). Information technology - security techniques - key management - part 4: Mechanisms based on weak secrets. Standard, International Organization for Standardization, Geneva, CH.

[105] Jablon, D. P. (1996). Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review*, 26(5):5–26.

[106] Jaspan, B. (1996). Dual-workfactor Encrypted Key Exchange: Efficiently Preventing Password Chaining and Dictionary Attacks. In *USENIX Security Symposium*.

[107] Kamara, S. and Papamanthou, C. (2013). Parallel and dynamic searchable symmetric encryption. In *International Conference on Financial Cryptography and Data Security*, pages 258–274. Springer.

[108] Kamara, S., Papamanthou, C., and Roeder, T. (2012). Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976.

[109] Katz, J. and Lindell, Y. (2014). *Introduction to modern cryptography*. CRC press.

[110] Katz, J., Ostrovsky, R., and Yung, M. (2001). Efficient password-authenticated key exchange using human-memorable passwords. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 475–494. Springer.

[111] Kemmerer, R. A. (1989). Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected areas in Communications*, 7(4):448–457.

[112] Kim, K. S., Kim, M., Lee, D., Park, J. H., and Kim, W.-H. (2017). Forward secure dynamic searchable symmetric encryption with efficient updates. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1449–1463. ACM.

[113] Kurosawa, K. and Ohtaki, Y. (2012). UC-secure searchable symmetric encryption. In *International Conference on Financial Cryptography and Data Security*, pages 285–298. Springer.

[114] Lai, S., Patranabis, S., Sakzad, A., Liu, J. K., Mukhopadhyay, D., Steinfeld, R., Sun, S.-F., Liu, D., and Zuo, C. (2018). Result pattern hiding searchable encryption for conjunctive queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 745–762.

[115] Lau, B., Chung, S., Song, C., Jang, Y., Lee, W., and Boldyreva, A. (2014). Mimesis aegis: A mimicry privacy shield–a system's approach to data privacy on public cloud. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 33–48.

[116] Lindell, Y. (2016). How to simulate it - A tutorial on the simulation proof technique. Technical report, IACR Cryptology ePrint Archive, 2016: 46.

[117] Liu, C., Zhu, L., Wang, M., and Tan, Y.-A. (2014). Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188.

[118] Lochbihler, A. (2016). Probabilistic functions and cryptographic oracles in higher order logic. In *European Symposium on Programming Languages and Systems*, pages 503–531. Springer.

[119] Lochbihler, A., Sefidgar, S. R., and Bhatt, B. (2017). Game-based cryptography in HOL. *Archive of Formal Proofs*, 2017.

[120] Lowe, G. (1996). Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer.

[121] Lowe, G. (1997). A hierarchy of authentication specifications. In *Computer security foundations workshop, 1997. Proceedings., 10th*, pages 31–43. IEEE.

[122] McHugh, J. (2001). An information flow tool for Gypsy. In *Seventeenth Annual Computer Security Applications Conference*, pages 191–201. IEEE.

[123] Meadows, C. (1996). The NRL protocol analyzer: An overview. *The Journal of Logic Programming*, 26(2):113–131.

[124] Meier, S., Cremers, C., and Basin, D. (2010). Strong invariants for the efficient construction of machine-checked protocol security proofs. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 231–245. IEEE.

[125] Meier, S., Schmidt, B., Cremers, C., and Basin, D. (2013). The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 696–701. Springer.

[126] Metere, R. and Dong, C. (2017). Automated cryptographic analysis of the Pedersen commitment scheme. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 275–287. Springer.

[127] Millen, J. and Shmatikov, V. (2001). Constraint solving for bounded-process cryptographic protocol analysis. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 166–175.

[128] Millen, J. K. (1995). The interrogator model. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 251–260. IEEE.

[129] Mitchell, J. C., Mitchell, M., and Stern, U. (1997). Automated analysis of cryptographic protocols using Mur,ϕ. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151. IEEE.

[130] Nanevski, A., Banerjee, A., and Garg, D. (2011). Verification of information flow and access control policies with dependent types. In *2011 IEEE Symposium on Security and Privacy*, pages 165–179. IEEE.

[131] Naor, M. (1991). Bit commitment using pseudorandomness. *Journal of cryptology*, 4(2):151–158.

[132] Naveed, M. (2015). The Fallacy of Composition of Oblivious RAM and Searchable Encryption. *IACR Cryptology ePrint Archive*, 2015:668.

[133] Naveed, M., Prabhakaran, M., and Gunter, C. A. (2014). Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy*, pages 639–654. IEEE.

[134] Needham, R. M. and Schroeder, M. D. (1978). Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999.

[135] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media.

[136] Pedersen, T. P. (1991). Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual International Cryptology Conference*, pages 129–140. Springer.

[137] Petcher, A. and Morrisett, G. (2015a). A Mechanized Proof of Security for Searchable Symmetric Encryption. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 481–494.

[138] Petcher, A. and Morrisett, G. (2015b). The foundational cryptography framework. In *International Conference on Principles of Security and Trust*, pages 53–72. Springer.

[139] Ramsdell, J. D. and Guttman, J. D. (2009). CPSA: A cryptographic protocol shapes analyzer. *Hackage. The MITRE Corporation*, 2(009).

[140] Rusinowitch, M. and Turuani, M. (2001). Protocol insecurity with finite number of sessions is NP-complete. *HAL Inria Archive*.

[141] Russello, G. (2018). ObliviousDB: Practical and Efficient Searchable Encryption with Controllable Leakage. In *Foundations and Practice of Security: 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers*, volume 10723, page 189. Springer.

[142] Ryan, M. D. and Smyth, B. (2011). Applied pi calculus. In Cortier, V. and Kremer, S., editors, *Formal Models and Techniques for Analyzing Security Protocols*. IOS Press.

[143] Ryan, P. and Schneider, S. A. (2001). *The modelling and analysis of security protocols: the csp approach*. Addison-Wesley Professional.

[144] Schmidt, B., Meier, S., Cremers, C., and Basin, D. (2012). Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 78–94. IEEE.

[145] Schnorr, C.-P. (1991). Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174.

[146] Schoenmakers, B. (1999). A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*, pages 148–164. Springer.

[147] Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715.

[148] Shoup, V. (2004). Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004:332.

[149] Smart, N. P. (2015). *Cryptography made simple*. Springer.

[150] Song, D. X., Berezin, S., and Perrig, A. (2001). Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1-2):47–74.

[151] Song, X., Dong, C., Yuan, D., Xu, Q., and Zhao, M. (2018). Forward private searchable symmetric encryption with optimized I/O efficiency. *IEEE Transactions on Dependable and Secure Computing*.

[152] Stefanov, E., Papamanthou, C., and Shi, E. (2014). Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, volume 71, pages 72–75.

[153] Stinson, D. R. (2005). *Cryptography: theory and practice*. CRC press.

[154] Stoughton, A. and Varia, M. (2017). Mechanizing the Proof of Adaptive, Information-theoretic Security of Cryptographic Protocols in the Random Oracle Model. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 83–99. IEEE.

[155] Swamy, N., Chen, J., and Chugh, R. (2010). Enforcing stateful authorization and information flow policies in Fine. In *European Symposium on Programming*, pages 529–549. Springer.

[156] Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., and Yang, J. (2011). Secure distributed programming with value-dependent types. In *ACM SIGPLAN Notices*, volume 46, pages 266–278. ACM.

[157] Tang, Q. and Mitchell, C. (2005). On the security of some password-based key agreement schemes. *Computational Intelligence and Security*, pages 149–154.

[158] Turuani, M. (2006). The CL-Atse protocol analyser. In *International Conference on Rewriting Techniques and Applications*, pages 277–286. Springer.

[159] Van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P., and Jonker, W. (2010). Computationally efficient searchable symmetric encryption. In *Workshop on Secure Data Management*, pages 87–100. Springer.

[160] Viganò, L. (2006). Automated security protocol analysis with the AVISPA tool. *Electronic Notes in Theoretical Computer Science*, 155:61–86.

[161] Woo, T. Y. and Lam, S. S. (1993). A semantic model for authentication protocols. In *Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on*, pages 178–194. IEEE.

[162] Yao, A. C. (1982). Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE.

[163] Zhang, M. (2004). Analysis of the SPEKE password-authenticated key exchange protocol. *IEEE Communications Letters*, 8(1):63–65.

[164] Zhang, Y., Katz, J., and Papamanthou, C. (2016). All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, Austin, TX. USENIX Association.

[165] Zheng, L. and Myers, A. C. (2007). Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84.

Appendix A

Cryptography

A.1 A popular example of secure computation: the two millionaires

The first and probably most popular example of a secure computation protocol is the *two millionaires*' problem [162], formulated almost 30 years ago by A. Yao and also known as Yao's millionaires' problem. Its main objective is very simple: two millionaires wish to determine who is richer. The formal description of such a problem simplifies the amount of wealth by a natural number, reducing the problem to determine which number is greater than the other. A simple way to solve it could be to first share and then compare their wealth. But additionally, they also do not want to reveal their actual wealth to the other millionaire. Therefore the solution above needs to be rethought in light of this additional, but not less important, privacy requirement.

Different solutions to this problem have been proposed, all based on different cryptographic primitives; however, we do not discuss them, as our interest is in its generalisation as all secure computation protocols.

A.2 Symmetric and asymmetric cryptography

Encryption systems are the core of cryptography. An encryption system is used when we want to protect the confidentiality of communication. For example, Alice wants to send the message m to Bob through an insecure channel in a way that only Bob can understand it. Encryption systems aim to solve this problem requiring Alice and Bob to hold some relating secrets, e.g. they both hold the same secret key k, that Alice can use to *encrypt* the message before sending it through the channel, and that Bob can use to *decrypt* and recover the original message.

A.2.1 Symmetric Encryption

Symmetric encryption is based on Alice and Bob knowing a secret key k before the communication. For this reason, symmetric encryption is also called private-

key encryption. Secret keys can be generated by a probabilistic efficient algorithm. Throughout this dissertation, we consider an algorithm *efficient* if it can be described as a probabilistic polynomial-time (PPT) Turing machine, i.e. it runs in time polynomial in n, where n is called the security parameter.. This is inline with the definition of Goldreich [85] but we do not introduce the formality typical of complexity theory as not necessary for our discussion. We now define symmetric encryption, borrowing the definition of symmetric encryption system from Katz and Lindell [109] using the shorter notation of Bellare et al. [33] as we find it more convenient.

Definition A.2.1 (Symmetric encryption)**.** *A* private-key encryption scheme *is a tuple of probabilistic polynomial-time algorithms* $(\mathsf{keygen}, \mathcal{E}, \mathcal{D})$ *such that:*

1. *The* key-generation algorithm keygen *takes as input* 1^n *(i.e. the security parameter written in unary) and outputs a key* k*; we write* $k \xleftarrow{\$} \mathsf{keygen}\,(1^n)$ *(emphasising that* keygen *is a randomized algorithm). We assume without loss of generality that any key output by* keygen (1^n) *satisfies* $|k| \geq n$*.*

2. *The* encryption algorithm $\mathcal{E}$ *takes as input a key* k *and a plaintext message* $m \in \{0,1\}^\star$*, and outputs a ciphertext* c*. Since* $\mathcal{E}$ *may be randomized, we write this as* $c \xleftarrow{\$} \mathcal{E}_k\,(m)$*.*

3. *The* decryption algorithm $\mathcal{D}$ *takes as input a key* k *and a ciphertext* c*, and outputs a message* m *or an error. We assume that* $\mathcal{D}$ *is deterministic, and so write* $m \leftarrow \mathcal{D}_k\,(c)$ *(assuming here that* $\mathcal{D}$ *does not return an error). We denote a generic error by the symbol* $\perp$*.*

It is required that for every n*, every* k *output by* keygen (1^n)*, and every* $m \in \{0,1\}^\star$*, it holds that* $\mathcal{D}_k\,(\mathcal{E}_k\,(m)) = m$*.*

If $(\mathsf{keygen}, \mathcal{E}, \mathcal{D})$ *is such that for* k *output by* keygen (1^n)*, algorithm* $\mathcal{E}_k$ *is only defined for messages* $m \in \{0,1\}^{\ell(n)}$*, then we say that* $(\mathsf{keygen}, \mathcal{E}, \mathcal{D})$ *is a* fixed-length private encryption scheme for messages of length $\ell\,(n)$.

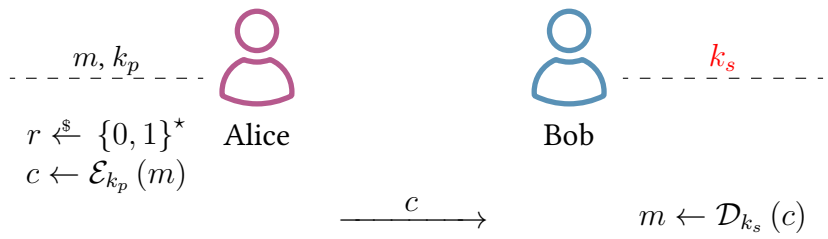
So when Alice wants to communicate, she does not send m directly, but rather sends its (probabilistic) encryption $c \xleftarrow{\$} \mathcal{E}_k\,(m)$. This way, Bob can decrypt it and recover the original message, $m \leftarrow \mathcal{D}_k\,(c)$. The confidentiality of the message m is therefore subject to the secrecy of the key k and the difficulty of reconstructing m by the adversary from inspecting ciphertexts.

A.2.2 Asymmetric encryption

A revolutionary point of view in the field cryptography was introduced in 1976, when Diffie and Hellman [75] shown to public for the first time the so called Diffie-Hellman key exchange introducing *asymmetric cryptography*. The novelty of asymmetric cryptography is that the key for the encryption algorithm can be publicly disclosed, so the sender needs no secrets to create the encrypted message. Yet, the encrypted message can only be decrypted by the owner of the corresponding private key. For this reason asymmetric cryptography is also called public-key cryptography.

With the goal of protecting the confidentiality of communication, Alice and Bob could employ a protocol based on an asymmetric cryptosystem, as illustrated in Figure A.1. In a preliminary phase the distribution of keys take place (securely), so

Fig. A.1 Protocol based on asymmetric encryption. It assumes that the keys have been properly and already distributed.



that Bob holds a secret key k_s whose corresponding public key k_p is provided to Alice. At this point, Alice can encrypt the message m with k_p and send the ciphertext $c \leftarrow \mathcal{E}_{k_p}(m)$ to Bob. The key point of the security in for a probabilistic asymmetric cryptosystem is that the knowledge of k_p and c is not enough to efficiently reconstruct the original message m.

Formally we describe a probabilistic asymmetric encryption through the Definition A.2.2. Throughout this chapter, we borrow our definitions from Katz and Lindell [109], with the notation of Bellare et al. [34].

Definition A.2.2 (Probabilistic asymmetric encryption)**.** *A* public-key encryption scheme *is a triple of probabilistic polynomial-time algorithms* $(\mathsf{keygen}, \mathcal{E}, \mathcal{D})$ *such that:*

1. *The* key-generation algorithm keygen *takes as input the security parameter* 1^n *and outputs a pair of keys* (k_p, k_s)*. We refer to the firs of these as the* public key *and the second as the* private key*. We assume for convenience that* k_p *and* k_s *each has length at least* n*, and that* n *can be determined from* k_p*,* k_s*.*

2. *The* encryption algorithm $\mathcal{E}$ *takes as input a public key* k_p *and a messsage* m *from some message space (that may depend on* k_p*). It outputs a cyphertext* c*, and we write this as* $c \xleftarrow{\$} \mathcal{E}_{k_p}(m)$*. (Looking ahead,* $\mathcal{E}$ *will need to be probabilistic to achieve meaningful security.)*

3. *The deterministic* decryption algorithm $\mathcal{D}$ *takes as input a private key* k_s *and a ciphertext* c*, and outputs a message* m *or a special symbol* $\perp$ *denoting failure. We write this as* $m \leftarrow \mathcal{D}_{k_s}(c)$*.*

It is required that, except possibly with negligible probability over (k_p, k_s) *output by* $\mathsf{keygen}(1^n)$*, we have* $\mathcal{D}_{k_s}\left(\mathcal{E}_{k_p}(m)\right) = m$ *for any (legal) message* m*.*

A function $\mu : \mathbb{N} \to \mathbb{R}$ is negligible if for every constant $c > 0$ there exists an integer n_c such that $\mu(n) \leq n^{-c}$ for all $n \geq n_c$.

The validity of a key pair strongly relates to the recovering of the plaintext from its corresponding ciphertext, but additional properties may be required, e.g. the length of the keys that in our definition is assumed to be a least n for simplicity.

The advantages brought by asymmetric encryption over symmetric encryption are three. The first advantage is that key distribution (Alice needs to be sure that k_p is Bob's) does not need a secure channel, and an authenticated channel suffices (e.g. Needham-Shroeder protocol). The second advantage is that the amount of key to store is reduced. For receiving messages, a party needs only one secret key, while all other parties use the corresponding public key. For sending messages, a public key for every entity is required; but differently from symmetric encryption, it is public information that can be centrally (and publicly) stored somewhere, allowing for non-expensive on-demand retrieval. The last advantage is that it allows two parties who had not previous interaction to communicate securely. This is a particularly important aspect that permeated the digital world and the Internet, e.g. a customer can buy online from a seller securely from a very long distance.

A.3 Symbolic vs Computational

To better appreciate the difference of reasoning between the symbolic model and the computational model, introduced in Section 2.3, we give a step-by-step formalisation of a generic two-party asymmetric encryption system. We introduced its definitions and concepts in Section A.2.

On the basis of the Definition A.2.2, the descriptions in the two models can be manifold, depending on the aspects of interest. We provide a fairly introductory description in the two models and compare them, showing similarities and dissimilarities. We remark that our interpretation is enough to capture the confidentiality of the message in both models; different interpretations may cover additional security properties.

A.3.1 A symbolic interpretation of asymmetric encryption

We refer to the formality of the applied pi calculus. The first step in the formal description are the basic elements defining the cryptosystem: sets, functions, theories and algorithms. Sets are commonly modelled and referred to as types. Functions are operations or predicates over those types and can be either deterministic or probabilistic. Theories determine the mathematical properties of sets and functions. Finally, algorithms are stateful or stateless processes: a process is a sequence of operations, they may return a value and can call other algorithms as sub-processes.

In the symbolic model, elements of sets are algebra terms and (initially) have only the equality property. To have more algebraic properties, one must explicitly add equations; the only equalities considered are those explicitly given. Some tools provide built-in equalities for popular sets like group theory, integer, pairing, and other theories; nevertheless they do not model the whole underlying mathematical theories. The behaviour of functions is modelled through rules and equations, that must be manually provided to capture the desired properties; otherwise, functions are considered as one-way functions. Theories are restricted to the explicit equations provided, so any implicit relation between algebra terms are ruled out from the rea-

soning as they are considered irrelevant (or redundant)[1]. Noticeably in the symbolic model, given the two terms c and $\mathcal{E}_{k_p}(m, r)$ (both ciphertexts), they never equal, if no rules or equations explicitly allow for it. Finally, algorithms, called processes, are seen as sequences of statements, with assignments, conditionals, network I/O operations, and calls to functions. Recursive functions may lead to non-terminating expansion, as they may expand forever. Other forms of loops are absent, as they would be *absorbed* by a single term and assumed to always terminate. They may be split into multiple sub-processes. The key generation algorithm keygen would not be explicitly called in the symbolic model, as the complexity aspects (first argument) are not captured. Rather, the randomness (second argument) would be captured by a fresh sampling of the secret key k_s, and k_p is commonly the result of a one-way function, say PK, applied to k_s. In pseudocode,

$$K = k_s \xleftarrow{\$} K_s; k_p \leftarrow \mathrm{PK}(k_s); \mathrm{send}(k_p)$$

where $\xleftarrow{\$}$ denote probabilistic assignment, and to remark that the public key is public domain, k_p is published in clear. As encryption is assumed to be perfect, keys are always valid; hence, the above description does not break the confidentiality of k_s, equivalently to calling keygen to generate both[2]. Even if we assume that the key distribution phase already took place, the model needs to formalise the key generation. Later in the model, we use the key in a way that implicitly captures the secure key distribution phase.

Summing up, apart from definitions, the model will have the equation to recover the plaintext from the ciphertext and the process P describing the protocol. Following the Definition A.2.2, we would model the equation

$$\forall k_s\ r\ m,\ \mathcal{D}_{k_s}\left(\mathcal{E}_{\mathrm{PK}(k_s)}(m, r)\right) = m, \tag{A.1}$$

that reconstructs the original message. A process K is called before the protocol itself to make sure that the keys are correctly generated and exchanged. Finally, we would describe the protocol in Figure A.1 with the processes:

$$\begin{aligned} A &= r \xleftarrow{\$} \{0,1\}^\star; c \leftarrow \mathcal{E}_{k_p}(m, r); \mathrm{send}(c) \\ B &= \mathrm{receive}(c); m \leftarrow \mathcal{D}_{k_s}(c) \\ P &= K; !(A|B) \end{aligned} \tag{A.2}$$

where A is (supposedly) run by Alice, and B by Bob, r is some randomness freshly sampled from $\{0,1\}^\star$, and P runs K at the beginning followed by infinite executions, denoted as !, of the concurrent execution, denoted as $|$, of A and B.

Very importantly, if we instantiate the encryption system to follow the practical construction of Rivest-Shamir-Adleman (RSA), the symbolic model would *not* change[3].

[1] This is one of the key aspects that allow the symbolic model to be more suitable for automation than the computational model.

[2] A freshly sampled value is always secret until published or reconstructed.

[3] One might pick up different and more meaningful naming for functions, but the final result would be the same, as perfect encryption is assumed.

Before illustrating the security property of confidentiality of the message in the symbolic model, we introduce a computational interpretation of the same protocol to compare with.

A.3.2 A computational interpretation of asymmetric encryption

In the computational model, sets are generally modelled by specific mathematical set theories, with all their definitions, propositions, and lemmas. For example, the theory of finite bit strings can model the set $\{0,1\}^n$. Functions are modelled through their mathematical definitions and theorems. Theories collect the definitions and the lemmas of sets and functions. Finally, algorithms are modelled as either functional or code-based. The former would adopt a single function per algorithm (this is the approach of CryptHol [32]); the latter follows an imperative style with a sequence of statements, including assignments, conditionals, loops, and function calls (this is the approach of EasyCrypt [26]). We adopt the code-based approach here as closer to programming languages and pseudocode.

Differently from the symbolic model, complexity and probability about algorithms are the most analysed aspects, so the key generation algorithm keygen is nothing especial: the keys are simply the output of the algorithm. The definitions of the asymmetric scheme can be formalised as in Algorithm 14.

Alg. 14. A computational model description of asymmetric encryption.

Π:
proc keygen $(n, r) : K$
proc $\mathcal{E}\,(k_p, m, r)$: C
proc $\mathcal{D}\,(k_s, c)$: P

Protocol:
proc keyDistrib ():
  $r \xleftarrow{\$} \{0,1\}^\star$
  $(\text{Bob}.k_s, \text{Alice}.k_p) \leftarrow$ keygen $(1^n, r)$
proc run (m):
  $\text{Alice}.m \leftarrow m$
  $c \leftarrow \text{Alice.send}\,()$
  $\text{Bob.receive}\,(c)$

Alice:
$m \in P$
$k_p \in KP$
proc send ():
  $r \xleftarrow{\$} \{0,1\}^\star$
  $c \leftarrow \mathcal{E}\,(k_p, m, r)$
  return c

Bob:
$m' \in P$
$k_s \in KS$
proc receive (c):
  $m' \leftarrow \mathcal{D}\,(k_s, c)$

Additionally, an (abstract) operator V to test the validity of the generated keys k_s and k_p must be provided to rule out invalid keys, $V\,(k_s, k_p)$. Differently from the symbolic model, the reconstruction property does not need special attention and is exactly the equation in Definition A.2.2. Furthermore, that equation will not be the only one considered throughout the reasoning: any other implicit equation that could be (even in principle) evinced from the theories would be considered.

Not assuming perfect cryptography, the algorithms in the computational model are suitable to describe real-world algorithms in details. So if we *specialised* the asymmetric encryption, as modelled in Algorithm 14, with the RSA cryptosystem, then we could include mathematical theories and construction details. The construction is illustrated in Algorithm 15 for completeness, but we do not discuss in detail as it would be out of our scope.

Alg. 15. A computational model of asymmetric encryption instantiated with the RSA cryptosystem.

RSA:
$N \in \mathbb{N}$
$C \equiv \mathbb{Z}_N^\star \times \{0,1\}^\star$
$\mathbb{P}$, set of prime numbers
$\gcd : \mathbb{N}_0 \times \mathbb{N}_0 \to \{\mathtt{T}, \mathtt{F}\}$, greatest common divisor
$\oplus : \mathbb{Z}_N^\star \times \{0,1\}^\star \to \mathbb{Z}_N^\star$, XOR with randomness
$\lambda : \mathbb{N}_0 \to \mathbb{N}_0$, totient function
$KP \equiv KS \equiv \mathbb{Z}_{\lambda(N)}^\star$
$C \equiv P \equiv \mathbb{Z}_N^\star$

proc keygen (n, r)**:**
- $p \xleftarrow{\$} \mathbb{P}$
- $q \xleftarrow{\$} \mathbb{P}$
- $N \leftarrow pq$ `// relating to` n`, e.g.` $N > 2^n$
- $e \xleftarrow{\$} \{e \in \mathbb{N} \mid 1 < e < \lambda(N) \wedge \gcd(e, \lambda(N)) = 1\}$
- $d \leftarrow e^{-1} \mod \lambda(N)$
- **return** (d, e)

proc $\mathcal{E}$ (k_p, m, r)**:**
- $c \leftarrow (m \oplus r)^{k_p} \mod N$
- **return** c

proc $\mathcal{D}$ (k_s, c)**:**
- $m \leftarrow c^{k_s} \mod N$
- **return** m

A.3.3 Reasoning about security properties

Desirable security properties vary from a protocol to another, but are generally described upon the concepts of *execution traces* and of *indistinguishability*.

Execution traces are the most peculiar to the symbolic model: a security property holds if it does for all traces of each run of the protocol in concurrent executions. A trace can be seen as a growing list of values generated by the execution of the protocol and partly available to the adversary. The computational model would let the same property hold except for a negligible number of traces.

Indistinguishability is the most peculiar concept to the computational model: a security property holds if any adversary can distinguish the execution of two processes by at most a negligible probability. The symbolic model denotes this notion as *process equivalence*, and their proof are more difficult to automate than properties on single traces, so that the most equivalence proofs are still manual [47].

We now refer to our example of the generic asymmetric encryption scheme, informally illustrated in Figure A.1, formally defined in Definition A.2.2 and formally modelled in symbolic in Equation A.2 and in computational in Algorithm 14. Here, the desirable security property is the **confidentiality** of the message sent by Alice to Bob.

In the symbolic model, confidentiality of m can be captured as a predicate on traces. We denote the set of all traces of the execution of $\Pi = (\mathsf{keygen}, \mathcal{E}, \mathcal{D})$ as T_Π, and the adversary's knowledge as $K_\mathcal{A}(t)$ where t is a given trace. Given the trace t, $K_\mathcal{A}(t)$ is a set containing all the terms that the adversary can infer by composing all the equations, functions, and values that are available, recursively. The predicate that capture confidentiality of m is the following:

$$\forall t \in T_\Pi,\ m \notin K_\mathcal{A}(t)\,. \tag{A.3}$$

In other words, m must not be reachable by the adversary's knowledge.

We omit a more formal description, as it would require us to report the whole semantics behind the inference rules, by which the number of traces can explode to infinite. A complete formal description can be read in the work from Cortier et al. [67], where they show how two notions of secrecy, described upon either *traces* or *indistinguishability*, relate. The former captures *syntactic secrecy*, where the adversary cannot reconstruct the full confidential data; the latter captures a stronger notion of secrect, *strong secrecy* [67], where the adversary has no information at all about the confidential data.

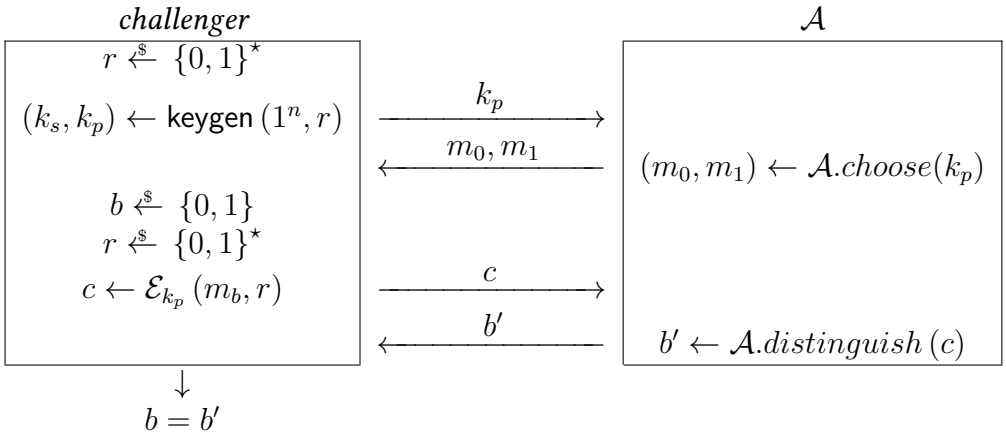
In the computational model, we would capture strong secrecy with an argument on indistinguishability: informally, an encrypted message should be indistinguishable from another ciphertext. To do so, we define a cryptographic experiment of indistinguishability, PubK illustrated in Figure A.2. The adversary of this experiment is called a *distinguisher*, denoted as $\mathcal{A}$, which is an abstract probabilistic polynomial Turing machine without a prescribed strategy, but with at least two procedures:

- $choose\,(k_p)$ that is given the key k_p used for encryption and outputs two (equal-length) messages m_0 and m_1; and
- $distinguish\,(c)$ that is given a ciphertext c and outputs a boolean value to relate back to either m_0 or m_1.

The experiment challenges the distinguisher by sending one of the messages she chooses, then checks whether the distinguisher correctly guesses which of the two messages was encrypted. The description of the adversary and the security experiment in the computational model are formalised in Algorithm 16. We notice that the adversary is an eavesdropper by the fact that the only data provided to *distinguish* is the ciphertext.

With the formal definitions of our security experiment in Algorithm 16, we can define security in terms of the probability of the experiment returning 1, i.e. the adversary is able to distinguish, non-negligibly better than a coin toss.

Fig. A.2 Flow of the experiment for the indistinguishability of encryptions of a probabilistic asymmetric encryption.



Alg. 16. Computational model interpretation for indistinguishability of ciphertexts for a public-key cryptosystem against an eavesdropper $\mathcal{A}$.

$\underline{\mathcal{A}}$: `// adversary`
proc *choose*$(k_s) : P \times P$ `// plaintexts`
proc *distinguish*$(c) : \{0,1\}$

$\underline{\mathsf{PubK}_{\Pi,\mathcal{A}}}$:
proc *main*():
  $r \xleftarrow{\$} \{0,1\}^\star$
  $(k_s, k_p) \leftarrow \mathsf{keygen}\,(1^n, r)$
  $(m_0, m_1) \leftarrow \mathcal{A}.choose\,(k_p)$
  $b \xleftarrow{\$} \{0,1\}$
  $r \xleftarrow{\$} \{0,1\}^\star$
  $c \leftarrow \mathcal{E}_{k_p}\,(m_b, r)$
  $b' \leftarrow \mathcal{A}.distinguish\,(c)$
  return $b = b'$

Definition A.3.1 (Indistinguishable encryptions in the presence of an eavesdropper)**.** *An probabilistic asymmetric encryption scheme* $\Pi = (\mathsf{keygen}, \mathcal{E}, \mathcal{D})$ *is said secure in terms of having* indistinguishable encryptions in the presence of an eavesdropper *if*

$$\forall \mathcal{A},\ \exists \mu,\ \mathsf{Pr}\left[\mathsf{PubK}_{\Pi,\mathcal{A}}\left(n\right) = 1\right] \leq \frac{1}{2} + \mu\left(n\right),$$

where μ *is a negligible function on the security parameter* n*.*

As the reader may appreciate, the interpretation in the computational model, even for a basic primitive as the asymmetric encryption, adds more complexity in the definitions and more details in the properties to prove. This complexity is reflected to the actual proofs and their automation by tools; unfortunately, the symbolic model, which can be highly automated, cannot capture all arguments in cryptography, nor in the same depth as the computational model. For this reason, the computational model offers stronger security guarantees.

We stress that our two models of the probabilistic asymmetric cryptosystem are simplified with the aim to highlight the *typical* reasoning on the symbolic and the computational models. Also, they are not the only possible interpretation, nor the most comprehensive in terms of security properties (we basically cover only security against chosen-plaintext attacks).

As a final remark, we emphasise that the two models show some similarities that sometimes overlap; in other words, in some cases the symbolic model may provide the same guarantees as the computational model. Nevertheless, we do not cover such overlap, and we refer to the seminal work of Abadi et al. [6] and the more recent survey from Blanchet [44] on techniques to *port* symbolic results to computational equivalents. For example, strong secrecy can be captured by the symbolic model in some cases [44], but we used the definition based on execution traces as it is the most common type of definition for security properties in the symbolic model.

Appendix B

About the proof of Sophos

B.1 Flaws in the original proof

B.1.1 The extract of the original proof

During our mechanisation of the proof of security by Bost [52] of the protocol $\Sigma o\varphi o\varsigma$, we encountered imprecisions in the proof steps that lead us to change the construction of the games to patch the proof. In particular, we refer to the equivalence between the games G_1 and $\widetilde{G_2}$ that are reported in Algorithms 17 and 18 respectively. This was not the only mistake in the original proof, but definitely the most interesting.

We quickly report a legend of their notation in Table B.1, as it is slightly different from ours.

Table B.1 Notation in the original construction of $\Sigma o\varphi o\varsigma$ and its proof of forward secrecy.

SK, PK	a secret key (the trapdoor) and a public key
W	table storing search tokens
T	table storing encrypted indexes
Key	table lazily filled by a random function
H_1	table lazily filled by a random function for the hash function H_1
UT	table lazily filled by a random function backing up hash values (for H_1)
ST, UT	search token and update token
ind	document index
π	trapdoor permutation

In their proof they argue:

> The point of $\widetilde{G_2}$ is to ensure consistency of H_1's transcript: in $\widetilde{G_2}$, H_1 is never programmed to two different values for the same input by Search' line 7. Instead of immediately generating the UT derived from the c-th ST for

Alg. 17. The game G_1 in the proof of security of $\Sigma o\varphi o\varsigma$ [52]. This is an adaptation from the construction of the real protocol after the pseudo-random function has been replaced by the random function whose values are memorised in the table Key.

Setup()

1: $(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
2: $\mathbf{W}, \mathbf{T} \leftarrow$ empty map
3: **return** $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$

Search$(w, \sigma; \mathsf{EDB})$

Client:
1: $K_w \leftarrow \mathtt{Key}[w]$
2: $(ST_c, c) \leftarrow \mathbf{W}[w]$ ▷ $c = n_w - 1$
3: **if** $(ST_c, c) = \bot$
4: **return** $\emptyset$
5: Send (K_w, ST_c, c) to the server.
Server:
6: **for** $i = c$ **to** 0 **do**
7: $UT_i \leftarrow H_1(K_w, ST_i)$
8: $e \leftarrow \mathbf{T}[UT_i]$
9: $\mathsf{ind} \leftarrow e \oplus H_2(K_w, ST_i)$
10: Output each ind
11: $ST_{i-1} \leftarrow \pi_{\mathsf{PK}}(ST_i)$
12: **end for**

Update$(\mathsf{add}, w, \mathsf{ind}, \sigma; \mathsf{EDB})$

Client:
1: $K_w \leftarrow \mathtt{Key}[w]$
2: $(ST_c, c) \leftarrow \mathbf{W}[w]$
3: **if** $(ST_c, c) = \bot$ **then**
4: $ST_0 \xleftarrow{\$} \mathcal{M}$, $c \leftarrow -1$
5: **else**
6: $ST_{c+1} \leftarrow \pi^{-1}_{\mathsf{SK}}(ST_c)$
7: **end if**
8: $\mathbf{W}[w] \leftarrow (ST_{c+1}, c+1)$
9: $UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
10: $e \leftarrow \mathsf{ind} \oplus H_2(K_w, ST_{c+1})$
11: Send (UT_{c+1}, e) to the server.
Server:
12: $\mathbf{T}[UT_{c+1}] \leftarrow e$

Alg. 18. The game $\widetilde{G_2}$ in the proof of security of $\Sigma o\varphi o\varsigma$. This is an extract from their proof [52].

Setup()

1: $(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
2: $\mathbf{W}, \mathbf{T} \leftarrow$ empty map
3: $\mathsf{bad} \leftarrow \mathsf{false}$
4: **return** $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$

Search$(w, \sigma; \mathsf{EDB})$

Client:
1: $K_w \leftarrow \mathtt{Key}[w]$
2: $(ST_0, \ldots, ST_c, c) \leftarrow \mathbf{W}[w]$
3: **if** $(ST_0, \ldots, ST_c, c) = \bot$
4: **return** $\emptyset$
5: $(\mathsf{ind}_0, \ldots, \mathsf{ind}_c) \leftarrow \mathsf{HistDB}(w)$ ▷ *Ordered from the order of updates*
6: **for** $i = 0$ **to** c **do**
7: $\mathtt{H}_1(K_w, ST_i) \leftarrow \mathtt{UT}[w, i]$
8: **end for**
9: Send (K_w, ST_c, c) to the server.
Server:
10: **for** $i = c$ **to** 0 **do**
11: $UT_i \leftarrow H_1(K_w, ST_i)$
12: $e \leftarrow \mathbf{T}[UT_i]$
13: $\mathsf{ind} \leftarrow e \oplus H_2(K_w, ST_i)$
14: Output each ind
15: $ST_{i-1} \leftarrow \pi_{\mathsf{PK}}(ST_i)$
16: **end for**

Update$(\mathsf{add}, w, \mathsf{ind}, \sigma; \mathsf{EDB})$

Client:
1: $K_w \leftarrow \mathtt{Key}[w]$
2: $(ST_0, \ldots, ST_c, c) \leftarrow \mathbf{W}[w]$
3: **if** $(ST_0, \ldots, ST_c, c) = \bot$ **then**
4: $ST_0 \xleftarrow{\$} \mathcal{M}$, $c \leftarrow -1$
5: **else**
6: $ST_{c+1} \leftarrow \pi^{-1}_{\mathsf{SK}}(ST_c)$
7: **end if**
8: $\mathbf{W}[w] \leftarrow (ST_0, \ldots, ST_{c+1}, c+1)$
9: $UT_{c+1} \leftarrow \{0,1\}^\mu$
10: **if** $\mathtt{H}_1(K_w, ST_{c+1}) \neq \bot$ **then**
11: $\mathsf{bad} \leftarrow \mathsf{true}$, $UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
12: **end if**
13: $\mathtt{UT}[w, c+1] \leftarrow UT_{c+1}$
14: $d \leftarrow \mathsf{ind} \oplus H_2(K_w, ST_{c+1})$
15: Send (UT_{c+1}, d) to the server.
Server:
16: $\mathbf{T}[UT_{c+1}] \leftarrow e$

$H_1(k, st)$

1: $v \leftarrow \mathtt{H}_1(k, st)$
2: **if** $v = \bot$ **then**
3: $v \xleftarrow{\$} \{0,1\}^\lambda$
4: **if** $\exists w, c$ s.t $st = ST_c \in \mathbf{W}[w]$ **then**
5: $\mathsf{bad} \leftarrow \mathsf{true}$, $v \leftarrow \mathtt{UT}[w, c]$
6: **end if**
7: $\mathtt{H}_1(k, st) \leftarrow v$
8: **end if**
9: **return** v

> keyword w from H_1, $\widetilde{G_2}$ randomly either choses them if (K_w, ST_{c+1}) does not already appear in H_1's transcript, or, if this is already the case, sets UT_{c+1} to the already chosen value $\mathsf{H}_1[K_w, UT_{c+1}]$. Then, $\widetilde{G_2}$ lazily programs the RO [random oracle] when needed by the Search protocol (line 7) or by an adversary's query (line 5 of H_1), so that it's outputs are consistent with the chosen values of the UT's.

And finally claim:

> Because of this, H_1's outputs in $\widetilde{G_2}$ and G_1 are perfectly indistinguishable, and so are the games:
>
> $$\mathsf{Pr}\left[\widetilde{G_2} = 1\right] = \mathsf{Pr}\left[G_1 = 1\right].$$

As we are about to show, both the argument and the claim are incorrect.

B.1.2 Flaws

The first flaw that can be spotted is in the simulation of the hash function. In particular, the guard at line 4 of H_1 should additionally test if $k = \mathsf{Key}[w]$, otherwise simply calling twice H_1 with different k would incorrectly raise the bad event. Interestingly enough, the authors must have noticed this problem, as their successive work [54] based on $\Sigma o\varphi o\varsigma$ fixes the guard.

Fixing this (minor?) mistake still is not enough to amend their argument, as other flaws can be spotted. In this on-paper proof, we could appreciate three important aspects that have been overlooked.

- some random functions, i.e. Key, are incorrectly treated as collision-free[1];
- permutations π are treated as cycle-free; and
- indistinguishability is entailed from the equivalence of a single procedure, H_1, rather than *all* procedures.

In the example we are about to show, where we demonstrate inconsistency issues in $\widetilde{G_2}$, we put ourselves in the situation when a collision in the random function occurs (with positive probability). We remark that collisions in the construction of $\Sigma o\varphi o\varsigma$ do not cause any problem; however, for the equivalence of the constructions $\widetilde{G_2}$ and G_1, collisions play a role that make $\widetilde{G_2}$ not be *perfectly indistinguishable* from G_1, thus breaking the proof. Fortunately, collisions are expected to be negligible events, so that a patch in this part of the proof can be made without breaking forward secrecy or throwing the whole proof away.

The claim we prove mistaken is the perfect indistinguishability of the two constructions $\widetilde{G_2}$ and G_1.

$$\mathsf{Pr}\left[\widetilde{G_2} = 1\right] = \mathsf{Pr}\left[G_1 = 1\right].$$

[1] Noticeably, other random functions, i.e. the hash functions, are correctly treated as *collision resistant*.

The above is equivalent to say that any situation that would allow a distinguisher to tell the two constructions apart must be impossible, i.e. the advantage must be 0.

$$\left|\mathsf{Pr}\left[\widetilde{G_2} = 1\right] - \mathsf{Pr}\left[G_1 = 1\right]\right| = 0.$$

We emphasise that this is not the only case where the inconsistency between the internal tables will break the perfect indistinguishability, yet it is enough to motivate the need for patching the original proof.

Operations leading to distinguish

The sequence of operations that allows a distinguisher to tell $\widetilde{G_2}$ and G_1 apart is very simple. Just after the setup (all tables are empty), two search operations on the keywords q and $q' \neq q$ are followed by two update operations over the same keywords. After the two search operations, the table Key contains two entries corresponding to q and q'.

The two updates called subsequently have a non-zero probability of breaking the consistency we need for later searches and calls to H_1. In particular, while the values in $\mathbf{Key}$ and $\mathbf{W}$ can be the same due to collisions when sampling twice from the same distribution, differently the pre-computed (actually sampled) hash values in $\mathbf{UT}$ can differ. With non-zero probability, we have that $\mathsf{Key}[q] = \mathsf{Key}[q']$ and $\mathsf{W}[q] = \mathsf{W}[q']$. If we focus on such a case, then the two constructions are easily distinguishable with the strategy described by the following steps:

Step 1 A call to search on q will output the values $\mathsf{Key}[q]$ and $\mathsf{W}[q]$. At the same time, the call will program the hash function H_1 with the value stored on $UT[q, 0]$ by the previous update in $\widetilde{G_2}$ and do nothing else in G_1.

Step 2 Now the adversary can call the hash function H_1 on $(\mathsf{Key}[q], \mathsf{W}[q])$ that will output the value $UT[q, 0]$ in $\widetilde{G_2}$, while in G_1 it will output the corresponding hash value.

Step 3 Similarly to the first step, a call to search on q' will output the values $\mathsf{Key}[q']$ and $\mathsf{W}[q']$, and at the same time will program the hash function H_1 with the value stored on $UT[q', 0]$.

Step 4 The adversary can call the hash function H_1 on $(\mathsf{Key}[q'], \mathsf{W}[q'])$ that will output the value $UT[q', 0]$.

Step 5 Finally, if the result from the hash function differ from the previous, then it must be interacting with the construction $\widetilde{G_2}$, as in G_1 it would never happen.

In detail in step 4, the adversary might be calling the hash function on the same values as step 2, as $\mathsf{Key}[q] = \mathsf{Key}[q']$ and $\mathsf{W}[q] = \mathsf{W}[q']$ has non-zero probability. If the adversary were interacting with $\widetilde{G_2}$, then she would certainly notice that H_1 on the same value unexpectedly produces two different digests[2]. On the contrary, if the adversary were interacting with G_1, the two digests would be the same.

[2]In such a case, the hash function H_1 would not even be a function!

This argument incontestably proves that $\widetilde{G}_2$ and G_1 are **not** perfectly indistinguishable and breaks the validity of the original proof of Sophos in [52].

B.2 The main game of the proof compared to the original

We did not need to completely change the original proof of Sophos provided by Bost [52]. The most of that proof is correct; however, they overlooked some steps that became flaws in their reasoning that we had to patch. The main sources of imprecisions are three:

- random functions are incorrectly treated as collision-free;
- permutations are incorrectly treated as cycle-free; and
- indistinguishability is incorrectly entailed by equivalence of a single procedure, rather than all procedures in the games.

The proof itself is complex and, while reading, the many details easily hide those steps. We noticed those problem because we tried and formalised the proof in the theorem prover EasyCrypt, where every gap must be filled. So, the structure of our proof is overall similar, but it differs on the important points already highlighted in Chapter 6. In particular, the main intermediate game of their proof, $\widetilde{G}_2$, is build in the classic fashion of bad events. In $\widetilde{G}_2$, all bad events are introduced while simulating oracle with random functions, and the construction keeps its internal consistency in the case bad events happen with the aim of being *perfectly* indistinguishable from the previous adjacent game in the proof. The original game $\widetilde{G}_2$ is illustrated in Algorithm 18 and their aim was to build it in such a way that

$$\mathsf{Pr}\left[\widetilde{G}_2 = 1\right] = \mathsf{Pr}\left[G_1 = 1\right].$$

As we showed in Appendix B.1, they did not manage to prove such equality, de facto invalidating their proof. So, we patched not only the mistakes in the pseudocode of the construction of Sophos and the simulator, but also the flaws in the indistinguishability proof of the sequence of games. While the former might be considered as minor mistakes or not (yet overlooked by the authors and the many reviewers[3]), the latter definitely invalidate the proof. By a quick look at the successive work based on [52], the same flaw is very likely to affect (some of) the proofs in Bost et al. [54], where the proof structure and claims are almost exactly the same.

The construction related to the main game of our mechanised proof is G_5, and it is illustrated in Algorithm 23. Differently from the main intermediate game $\widetilde{G}_2$ of the original proof (see Algorithm 18), in G_5 of our model we discard the bad events without keeping consistency. The reason behind our choice is that we found it very difficult to keep consistency with so many asynchronous tables and bad events. The

[3]Both [52] and [53] report the same imprecisions.

price we pay for that is that we double the probability of some of the negligible events: more details are discussed in Chapter 6. For completeness, we also report all the other intermediate games. The original construction, patched and formalised with our formalism, have been shown in Algorithm 10; similarly the simulator have been illustrated in Algorithm 12.

Alg. 19. The model of G_1 client in pseudo-code. The pseudo-random function F is substituted with a keyed random function kf.

$\underline{G_1.Client}$:

$k \in \{0,1\}^\lambda$
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha \times \mathbb{N}$
$kf : \{0,1\}^l \to \{0,1\}^l \to \{0,1\}^l$ `// keyed random function`

proc $\mathcal{I}_c(1^\lambda)$:
1 $k \xleftarrow{\$} \{0,1\}^\lambda$
2 $\alpha, \tau \leftarrow \mathcal{K}\left(1^\lambda\right)$
3 $W \leftarrow \emptyset$
4 **return** α

proc $\mathcal{S}_c(w)$:
5 **if** $w \in W$ **then**
6 $k_w \leftarrow kf\,(k, w)$
7 $s, c \leftarrow W\,[w]$
8 $r \leftarrow (k_w, s, c)$
9 **else**
10 $r \leftarrow \perp$
11 **return** r

proc $\mathcal{O}_c(i, k_w, s)$:
12 $h \leftarrow \perp$
13 **if** $i = H_1$ **then**
14 $h \leftarrow (H_1\,(k_w, s)\,, \perp)$
15 **else if** $i = H_2$ **then**
16 $h \leftarrow (\perp, H_2\,(k_w, s))$
17 **return** h

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
18 $k_w \leftarrow kf\,(k, w)$
19 **if** $w \notin W$ **then**
20 $s \xleftarrow{\$} D_\alpha$
21 $c \leftarrow 0$
22 **else**
23 $s, c \leftarrow W\,[w]$
24 $s \leftarrow \mathcal{B}_\tau\,(s)$
25 $c \leftarrow c + 1$
26 $W\,[w] \leftarrow (s, c)$
27 $t \leftarrow H_1\,(k_w, s)$
28 $e \leftarrow i \oplus H_2\,(k_w, s)$
29 **return** (t, e)

Alg. 20. The model of G_2 client in pseudo-code. The hash function H_1 is replaced with the random oracle h_1.

$\underline{G_2.Client}$:

$k \in \{0,1\}^\lambda$
$\tau \in T$ // trapdoor
$W : \{0,1\}^l \to D_\alpha \times \mathbb{N}$
$kf : \{0,1\}^l \to \{0,1\}^l \to \{0,1\}^l$
h_1 // random oracle for H1

proc $\mathcal{I}_c(1^\lambda)$:
1 $k \xleftarrow{\$} \{0,1\}^\lambda$
2 $\alpha, \tau \leftarrow \mathcal{K}\left(1^\lambda\right)$
3 $W \leftarrow \emptyset$
4 **return** α

proc $\mathcal{S}_c(w)$:
5 **if** $w \in W$ **then**
6 $k_w \leftarrow kf\,(k, w)$
7 $s, c \leftarrow W\,[w]$
8 $r \leftarrow (k_w, s, c)$
9 **else**
10 $r \leftarrow \perp$
11 **return** r

proc $\mathcal{O}_c(i, k_w, s)$:
12 $h \leftarrow \perp$
13 **if** $i = H_1$ **then**
14 $h \leftarrow (h_1\,(k_w, s)\,, \perp)$
15 **else if** $i = H_2$ **then**
16 $h \leftarrow (\perp, H_2\,(k_w, s))$
17 **return** h

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
18 $k_w \leftarrow kf\,(k, w)$
19 **if** $w \notin W$ **then**
20 $s \xleftarrow{\$} D_\alpha$
21 $c \leftarrow 0$
22 **else**
23 $s, c \leftarrow W\,[w]$
24 $s \leftarrow \mathcal{B}_\tau\,(s)$
25 $c \leftarrow c + 1$
26 $W\,[w] \leftarrow (s, c)$
27 $t \leftarrow h_1\,(k_w, s)$
28 $e \leftarrow i \oplus H_2\,(k_w, s)$
29 **return** (t, e)

Alg. 21. The model of G_3 client in pseudo-code. The hash function H_2 is replaced with the random oracle h_2.

$\underline{G_3.Client}$:

$k \in \{0,1\}^\lambda$
$\tau \in T$ // trapdoor
$W : \{0,1\}^l \to D_\alpha \times \mathbb{N}$
$kf : \{0,1\}^l \to \{0,1\}^l \to \{0,1\}^l$
h_1 // random oracle for H1
h_2 // random oracle for H2

proc $\mathcal{I}_c(1^\lambda)$:
1 $k \xleftarrow{\$} \{0,1\}^\lambda$
2 $\alpha, \tau \leftarrow \mathcal{K}\left(1^\lambda\right)$
3 $W \leftarrow \emptyset$
4 **return** α

proc $\mathcal{S}_c(w)$:
5 **if** $w \in W$ **then**
6 $k_w \leftarrow kf\,(k, w)$
7 $s, c \leftarrow W\,[w]$
8 $r \leftarrow (k_w, s, c)$
9 **else**
10 $r \leftarrow \perp$
11 **return** r

proc $\mathcal{O}_c(i, k_w, s)$:
12 $h \leftarrow \perp$
13 **if** $i = H_1$ **then**
14 $h \leftarrow (h_1\,(k_w, s)\,, \perp)$
15 **else if** $i = H_2$ **then**
16 $h \leftarrow (\perp, h_2\,(k_w, s))$
17 **return** h

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
18 $k_w \leftarrow kf\,(k, w)$
19 **if** $w \notin W$ **then**
20 $s \xleftarrow{\$} D_\alpha$
21 $c \leftarrow 0$
22 **else**
23 $s, c \leftarrow W\,[w]$
24 $s \leftarrow \mathcal{B}_\tau\,(s)$
25 $c \leftarrow c + 1$
26 $W\,[w] \leftarrow (s, c)$
27 $t \leftarrow h_1\,(k_w, s)$
28 $e \leftarrow i \oplus h_2\,(k_w, s)$
29 **return** (t, e)

Alg. 22. The model of G_4 client in pseudo-code. The keyed random function kf is replaced with the random function f_k.

$\underline{G_4.Client}$:

$k \in \{0,1\}^\lambda$
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha \times \mathbb{N}$
$f_k : \{0,1\}^l \to \{0,1\}^l$
h_1 `// random oracle for H1`
h_2 `// random oracle for H2`

proc $\mathcal{I}_c(1^\lambda)$:
1 $k \xleftarrow{\$} \{0,1\}^\lambda$
2 $\alpha, \tau \leftarrow \mathcal{K}\left(1^\lambda\right)$
3 $W \leftarrow \emptyset$
4 **return** α

proc $\mathcal{S}_c(w)$:
5 **if** $w \in W$ **then**
6 $k_w \leftarrow f_k(w)$
7 $s, c \leftarrow W[w]$
8 $r \leftarrow (k_w, s, c)$
9 **else**
10 $r \leftarrow \perp$
11 **return** r

proc $\mathcal{O}_c(i, k_w, s)$:
12 $h \leftarrow \perp$
13 **if** $i = H_1$ **then**
14 $h \leftarrow (h_1(k_w, s), \perp)$
15 **else if** $i = H_2$ **then**
16 $h \leftarrow (\perp, h_2(k_w, s))$
17 **return** h

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
18 $k_w \leftarrow f_k(w)$
19 **if** $w \notin W$ **then**
20 $s \xleftarrow{\$} D_\alpha$
21 $c \leftarrow 0$
22 **else**
23 $s, c \leftarrow W[w]$
24 $s \leftarrow \mathcal{B}_\tau(s)$
25 $c \leftarrow c + 1$
26 $W[w] \leftarrow (s, c)$
27 $t \leftarrow h_1(k_w, s)$
28 $e \leftarrow i \oplus h_2(k_w, s)$
29 **return** (t, e)

Alg. 23. The model of G_5 client in pseudo-code. Bad events are introduced as well as the simulation of hash functions.

$\underline{G_5.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ // random function
$\tau \in T$ // trapdoor
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$b_c, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, b_t \in \{0,1\}$

$\underline{\text{Client}}$:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
1 $k_w \leftarrow f(w)$
2 **if** f *has collisions* **then**
3 $b_c \leftarrow 1$
4 **return** $\perp$
5 **if** $w \notin W$ **then**
6 $s_c \leftarrow \emptyset$
7 $s \xleftarrow{\$} D_\alpha$
8 $c \leftarrow 0$
9 **else**
10 $s_c \leftarrow W[w]$
11 $s \leftarrow \mathcal{B}_\tau(s)$
12 $c \leftarrow |s_c|$
13 $W[w] \leftarrow s_c || [s]$
14 **if** $(k_w, s) \in T_{H_1}$ **then**
15 $b_{t_1} \leftarrow 1$
16 **return** $\perp$
17 **if** $(k_w, s) \in T_{H_2}$ **then**
18 $b_{t_2} \leftarrow 1$
19 **return** $\perp$
20 **if** $s \in s_c$ **then**
21 $b_t \leftarrow 1$
22 **return** $\perp$
23 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
24 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
25 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$:
26 $h \leftarrow \perp$
27 **if** $i = H_1$ **then**
28 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
29 **else if** $i = H_2$ **then**
30 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
31 **return** h

proc $\mathcal{S}_c(w)$:
32 **if** $w \in W$ **then**
33 $k_w \leftarrow f(w)$
34 **if** f *has collisions* **then**
35 $b_c \leftarrow 1$
36 **return** $\perp$
37 $s_c \leftarrow W[w]$
38 $c \leftarrow |s_c| - 1$
39 $r \leftarrow (k_w, s_c[c], c)$
40 $i \leftarrow 0$
41 **while** $i < |s_c|$ **do**
42 $s \leftarrow s_c[i]$
43 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w,i)]$
44 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w,i)]$
45 $i \leftarrow i + 1$
46 **else**
47 $r \leftarrow \perp$
48 **return** r

proc $\mathsf{SimH}_1(k_w, s)$:
49 **if** $(k_w, s) \notin T_{H_1}$ **then**
50 $w_s \leftarrow \{w \in f | f(w) = k_w\}$
51 $W' \leftarrow W|_{\{w \in W | w \in w_s \wedge s \in W[w]\}}$
52 **if** $W' \neq \emptyset$ **then**
53 $b_{h_1} \leftarrow 1$
54 **return** $\perp$
55 **else**
56 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
57 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$:
58 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$:
59 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
60 $W \leftarrow \emptyset$
61 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
62 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
63 $b_c \leftarrow b_t \leftarrow 0$
64 $b_{h_1} \leftarrow b_{h_2} \leftarrow b_{t_1} \leftarrow b_{t_2} \leftarrow 0$
65 **return** α

Alg. 24. The model of G_6 client in pseudo-code. Bad event b_{h_2} is not handled.

$\underline{G_6.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ // random function
$\tau \in T$ // trapdoor
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$b_c, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, b_t \in \{0,1\}$

$\underline{\text{Client}}$:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 $k_w \leftarrow f(w)$
2 **if** f *has collisions* **then**
3 $b_c \leftarrow 1$
4 **return** $\perp$
5 **if** $w \notin W$ **then**
6 $s_c \leftarrow \emptyset$
7 $s \xleftarrow{\$} D_\alpha$
8 $c \leftarrow 0$
9 **else**
10 $s_c \leftarrow W[w]$
11 $s \leftarrow \mathcal{B}_\tau(s)$
12 $c \leftarrow |s_c|$
13 $W[w] \leftarrow s_c || [s]$
14 **if** $(k_w, s) \in T_{H_1}$ **then**
15 $b_{t_1} \leftarrow 1$
16 **return** $\perp$
17 **if** $(k_w, s) \in T_{H_2}$ **then**
18 $b_{t_2} \leftarrow 1$
19 **return** $\perp$
20 **if** $s \in s_c$ **then**
21 $b_t \leftarrow 1$
22 **return** $\perp$
23 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
24 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
25 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
26 $h \leftarrow \perp$
27 **if** $i = H_1$ **then**
28 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
29 **else if** $i = H_2$ **then**
30 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
31 **return** h

proc $\mathcal{S}_c(w)$**:**
32 **if** $w \in W$ **then**
33 $k_w \leftarrow f(w)$
34 **if** f *has collisions* **then**
35 $b_c \leftarrow 1$
36 **return** $\perp$
37 $s_c \leftarrow W[w]$
38 $c \leftarrow |s_c| - 1$
39 $r \leftarrow (k_w, s_c[c], c)$
40 $i \leftarrow 0$
41 **while** $i < |s_c|$ **do**
42 $s \leftarrow s_c[i]$
43 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w,i)]$
44 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w,i)]$
45 $i \leftarrow i + 1$
46 **else**
47 $r \leftarrow \perp$
48 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
49 **if** $(k_w, s) \notin T_{H_1}$ **then**
50 $w_s \leftarrow \{w \in f | f(w) = k_w\}$
51 $W' \leftarrow W|_{\{w \in W | w \in w_s \wedge s \in W[w]\}}$
52 **if** $W' \neq \emptyset$ **then**
53 $b_{h_1} \leftarrow 1$
54 **return** $\perp$
55 **else**
56 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
57 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
58 **if** $(k_w, s) \notin T_{H_2}$ **then**
59 $T_{H_2}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
60 **return** $T_{H_2}[(k_w, s)]$

proc $\mathcal{I}_c(1^\lambda)$**:**
61 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
62 $W \leftarrow \emptyset$
63 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
64 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
65 $b_c \leftarrow b_t \leftarrow 0$
66 $b_{h_1} \leftarrow b_{h_2} \leftarrow b_{t_1} \leftarrow b_{t_2} \leftarrow 0$
67 **return** α

Alg. 25. The model of G_7 client in pseudo-code. Bad event b_{h_1} is not handled.

$\underline{G_7.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ // random function
$\tau \in T$ // trapdoor
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$b_c, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, b_t \in \{0,1\}$

$\underline{\text{Client}}$:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 $k_w \leftarrow f(w)$
2 **if** f *has collisions* **then**
3 $b_c \leftarrow 1$
4 **return** $\perp$
5 **if** $w \notin W$ **then**
6 $s_c \leftarrow \emptyset$
7 $s \xleftarrow{\$} D_\alpha$
8 $c \leftarrow 0$
9 **else**
10 $s_c \leftarrow W[w]$
11 $s \leftarrow \mathcal{B}_\tau(s)$
12 $c \leftarrow |s_c|$
13 $W[w] \leftarrow s_c || [s]$
14 **if** $(k_w, s) \in T_{H_1}$ **then**
15 $b_{t_1} \leftarrow 1$
16 **return** $\perp$
17 **if** $(k_w, s) \in T_{H_2}$ **then**
18 $b_{t_2} \leftarrow 1$
19 **return** $\perp$
20 **if** $s \in s_c$ **then**
21 $b_t \leftarrow 1$
22 **return** $\perp$
23 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
24 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
25 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
26 $h \leftarrow \perp$
27 **if** $i = H_1$ **then**
28 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
29 **else if** $i = H_2$ **then**
30 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
31 **return** h

proc $\mathcal{S}_c(w)$**:**
32 **if** $w \in W$ **then**
33 $k_w \leftarrow f(w)$
34 **if** f *has collisions* **then**
35 $b_c \leftarrow 1$
36 **return** $\perp$
37 $s_c \leftarrow W[w]$
38 $c \leftarrow |s_c| - 1$
39 $r \leftarrow (k_w, s_c[c], c)$
40 $i \leftarrow 0$
41 **while** $i < |s_c|$ **do**
42 $s \leftarrow s_c[i]$
43 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w,i)]$
44 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w,i)]$
45 $i \leftarrow i + 1$
46 **else**
47 $r \leftarrow \perp$
48 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
49 **if** $(k_w, s) \notin T_{H_1}$ **then**
50 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
51 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
52 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$**:**
53 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
54 $W \leftarrow \emptyset$
55 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
56 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
57 $b_c \leftarrow b_t \leftarrow 0$
58 $b_{h_1} \leftarrow b_{h_2} \leftarrow b_{t_1} \leftarrow b_{t_2} \leftarrow 0$
59 **return** α

Alg. 26. The model of G_8 client in pseudo-code. Bad event b_t is not handled.

$\underline{G_8.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ `// random function`
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$b_c, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, b_t \in \{0,1\}$

$\underline{\text{Client}}$:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 $k_w \leftarrow f(w)$
2 **if** f *has collisions* **then**
3 $b_c \leftarrow 1$
4 **return** $\perp$
5 **if** $w \notin W$ **then**
6 $s_c \leftarrow \emptyset$
7 $s \xleftarrow{\$} D_\alpha$
8 $c \leftarrow 0$
9 **else**
10 $s_c \leftarrow W[w]$
11 $s \leftarrow \mathcal{B}_\tau(s)$
12 $c \leftarrow |s_c|$
13 $W[w] \leftarrow s_c || [s]$
14 **if** $(k_w, s) \in T_{H_1}$ **then**
15 $b_{t_1} \leftarrow 1$
16 **return** $\perp$
17 **if** $(k_w, s) \in T_{H_2}$ **then**
18 $b_{t_2} \leftarrow 1$
19 **return** $\perp$
20 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
21 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
22 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
23 $h \leftarrow \perp$
24 **if** $i = H_1$ **then**
25 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
26 **else if** $i = H_2$ **then**
27 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
28 **return** h

proc $\mathcal{S}_c(w)$**:**
29 **if** $w \in W$ **then**
30 $k_w \leftarrow f(w)$
31 **if** f *has collisions* **then**
32 $b_c \leftarrow 1$
33 **return** $\perp$
34 $s_c \leftarrow W[w]$
35 $c \leftarrow |s_c| - 1$
36 $r \leftarrow (k_w, s_c[c], c)$
37 $i \leftarrow 0$
38 **while** $i < |s_c|$ **do**
39 $s \leftarrow s_c[i]$
40 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w, i)]$
41 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w, i)]$
42 $i \leftarrow i + 1$
43 **else**
44 $r \leftarrow \perp$
45 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
46 **if** $(k_w, s) \notin T_{H_1}$ **then**
47 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
48 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
49 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$**:**
50 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
51 $W \leftarrow \emptyset$
52 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
53 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
54 $b_c \leftarrow b_t \leftarrow 0$
55 $b_{h_1} \leftarrow b_{h_2} \leftarrow b_{t_1} \leftarrow b_{t_2} \leftarrow 0$
56 **return** α

Alg. 27. The model of G_9 client in pseudo-code. Bad event b_{t_2} is not handled.

$\underline{G_9.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ // random function
$\tau \in T$ // trapdoor
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$b_c, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, b_t \in \{0,1\}$

$\underline{\text{Client}}$:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
1 $k_w \leftarrow f(w)$
2 **if** f *has collisions* **then**
3 $b_c \leftarrow 1$
4 **return** $\perp$
5 **if** $w \notin W$ **then**
6 $s_c \leftarrow \emptyset$
7 $s \xleftarrow{\$} D_\alpha$
8 $c \leftarrow 0$
9 **else**
10 $s_c \leftarrow W[w]$
11 $s \leftarrow \mathcal{B}_\tau(s)$
12 $c \leftarrow |s_c|$
13 $W[w] \leftarrow s_c || [s]$
14 **if** $(k_w, s) \in T_{H_1}$ **then**
15 $b_{t_1} \leftarrow 1$
16 **return** $\perp$
17 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
18 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
19 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$:
20 $h \leftarrow \perp$
21 **if** $i = H_1$ **then**
22 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
23 **else if** $i = H_2$ **then**
24 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
25 **return** h

proc $\mathcal{S}_c(w)$:
26 **if** $w \in W$ **then**
27 $k_w \leftarrow f(w)$
28 **if** f *has collisions* **then**
29 $b_c \leftarrow 1$
30 **return** $\perp$
31 $s_c \leftarrow W[w]$
32 $c \leftarrow |s_c| - 1$
33 $r \leftarrow (k_w, s_c[c], c)$
34 $i \leftarrow 0$
35 **while** $i < |s_c|$ **do**
36 $s \leftarrow s_c[i]$
37 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w,i)]$
38 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w,i)]$
39 $i \leftarrow i + 1$
40 **else**
41 $r \leftarrow \perp$
42 **return** r

proc $\mathsf{SimH}_1(k_w, s)$:
43 **if** $(k_w, s) \notin T_{H_1}$ **then**
44 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
45 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$:
46 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$:
47 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
48 $W \leftarrow \emptyset$
49 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
50 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
51 $b_c \leftarrow b_t \leftarrow 0$
52 $b_{h_1} \leftarrow b_{h_2} \leftarrow b_{t_1} \leftarrow b_{t_2} \leftarrow 0$
53 **return** α

Alg. 28. The model of G_{10} client in pseudo-code. Bad event b_{t_1} is not handled.

$G_{10}.Client$:

$f : \{0,1\}^l \to \{0,1\}^l$ `// random function`
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$b_c, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, b_t \in \{0,1\}$

Client:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 $k_w \leftarrow f(w)$
2 **if** f *has collisions* **then**
3 $b_c \leftarrow 1$
4 **return** $\perp$
5 **if** $w \notin W$ **then**
6 $s_c \leftarrow \emptyset$
7 $s \xleftarrow{\$} D_\alpha$
8 $c \leftarrow 0$
9 **else**
10 $s_c \leftarrow W[w]$
11 $s \leftarrow \mathcal{B}_\tau(s)$
12 $c \leftarrow |s_c|$
13 $W[w] \leftarrow s_c || [s]$
14 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
15 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
16 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
17 $h \leftarrow \perp$
18 **if** $i = H_1$ **then**
19 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
20 **else if** $i = H_2$ **then**
21 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
22 **return** h

proc $\mathcal{S}_c(w)$**:**
23 **if** $w \in W$ **then**
24 $k_w \leftarrow f(w)$
25 **if** f *has collisions* **then**
26 $b_c \leftarrow 1$
27 **return** $\perp$
28 $s_c \leftarrow W[w]$
29 $c \leftarrow |s_c| - 1$
30 $r \leftarrow (k_w, s_c[c], c)$
31 $i \leftarrow 0$
32 **while** $i < |s_c|$ **do**
33 $s \leftarrow s_c[i]$
34 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w, i)]$
35 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w, i)]$
36 $i \leftarrow i + 1$
37 **else**
38 $r \leftarrow \perp$
39 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
40 **if** $(k_w, s) \notin T_{H_1}$ **then**
41 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
42 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
43 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$**:**
44 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
45 $W \leftarrow \emptyset$
46 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
47 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
48 $b_c \leftarrow b_t \leftarrow 0$
49 $b_{h_1} \leftarrow b_{h_2} \leftarrow b_{t_1} \leftarrow b_{t_2} \leftarrow 0$
50 **return** α

Alg. 29. The model of G_{11} client in pseudo-code. Bad event b_c is not handled.

$\underline{G_{11}.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ `// random function`
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$b_c, b_{h_1}, b_{h_2}, b_{t_1}, b_{t_2}, b_t \in \{0,1\}$

$\underline{\text{Client}}$:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
1 $k_w \leftarrow f(w)$
2 **if** $w \notin W$ **then**
3 $s_c \leftarrow \emptyset$
4 $s \xleftarrow{\$} D_\alpha$
5 $c \leftarrow 0$
6 **else**
7 $s_c \leftarrow W[w]$
8 $s \leftarrow \mathcal{B}_\tau(s)$
9 $c \leftarrow |s_c|$
10 $W[w] \leftarrow s_c || [s]$
11 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
12 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
13 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$:
14 $h \leftarrow \perp$
15 **if** $i = H_1$ **then**
16 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
17 **else if** $i = H_2$ **then**
18 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
19 **return** h

proc $\mathcal{S}_c(w)$:
20 **if** $w \in W$ **then**
21 $k_w \leftarrow f(w)$
22 $s_c \leftarrow W[w]$
23 $c \leftarrow |s_c| - 1$
24 $r \leftarrow (k_w, s_c[c], c)$
25 $i \leftarrow 0$
26 **while** $i < |s_c|$ **do**
27 $s \leftarrow s_c[i]$
28 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w, i)]$
29 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w, i)]$
30 $i \leftarrow i + 1$
31 **else**
32 $r \leftarrow \perp$
33 **return** r

proc $\mathsf{SimH}_1(k_w, s)$:
34 **if** $(k_w, s) \notin T_{H_1}$ **then**
35 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
36 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$:
37 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$:
38 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
39 $W \leftarrow \emptyset$
40 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
41 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
42 $b_c \leftarrow b_t \leftarrow 0$
43 $b_{h_1} \leftarrow b_{h_2} \leftarrow b_{t_1} \leftarrow b_{t_2} \leftarrow 0$
44 **return** α

Alg. 30. The model of G_{12} client in pseudo-code. Complete removal of bad events.

$G_{12}.Client$:

$f : \{0,1\}^l \to \{0,1\}^l$ `// random function`
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$

Client:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 $k_w \leftarrow f(w)$
2 **if** $w \notin W$ **then**
3 $s_c \leftarrow \emptyset$
4 $s \xleftarrow{\$} D_\alpha$
5 $c \leftarrow 0$
6 **else**
7 $s_c \leftarrow W[w]$
8 $s \leftarrow \mathcal{B}_\tau(s)$
9 $c \leftarrow |s_c|$
10 $W[w] \leftarrow s_c || [s]$
11 $T_t[(w,c)] \xleftarrow{\$} \{0,1\}^\lambda$
12 $T_e[(w,c)] \xleftarrow{\$} \{0,1\}^{\lambda'}$
13 **return** $(T_t[(w,c)], i \oplus T_e[(w,c)])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
14 $h \leftarrow \perp$
15 **if** $i = H_1$ **then**
16 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
17 **else if** $i = H_2$ **then**
18 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
19 **return** h

proc $\mathcal{S}_c(w)$**:**
20 **if** $w \in W$ **then**
21 $k_w \leftarrow f(w)$
22 $s_c \leftarrow W[w]$
23 $c \leftarrow |s_c| - 1$
24 $r \leftarrow (k_w, s_c[c], c)$
25 $i \leftarrow 0$
26 **while** $i < |s_c|$ **do**
27 $s \leftarrow s_c[i]$
28 $T_{H_1}[(k_w, s)] \leftarrow T_t[(w, i)]$
29 $T_{H_2}[(k_w, s)] \leftarrow T_e[(w, i)]$
30 $i \leftarrow i + 1$
31 **else**
32 $r \leftarrow \perp$
33 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
34 **if** $(k_w, s) \notin T_{H_1}$ **then**
35 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
36 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
37 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$**:**
38 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
39 $W \leftarrow \emptyset$
40 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
41 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
42 **return** α

Alg. 31. The model of G_{13} client in pseudo-code. Simulate results using update-access pattern.

$\underline{G_{13}.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ // random function
$\tau \in T$ // trapdoor
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \mathbb{N} \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \mathbb{N} \to \{0,1\}^{\lambda'}$
$t : \mathbb{N}$ // timestamp
$T : \{0,1\}^l \to (\mathbb{N} \times D_\alpha)^\star$

Client:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$:
1 $k_w \leftarrow f(w)$
2 **if** $w \notin W$ **then**
3 $s_c \leftarrow t_0 \leftarrow \emptyset$
4 $s \xleftarrow{\$} D_\alpha$
5 $c \leftarrow 0$
6 **else**
7 $t_0 \leftarrow T[t]$
8 $s_c \leftarrow W[w]$
9 $s \leftarrow \mathcal{B}_\tau(s)$
10 $c \leftarrow |s_c|$
11 $W[w] \leftarrow s_c || [s]$
12 $T[w] \leftarrow t_0 || [(t,i)]$
13 $T_t[t] \xleftarrow{\$} \{0,1\}^\lambda$
14 $T_e[t] \xleftarrow{\$} \{0,1\}^{\lambda'}$
15 $t \leftarrow t+1$
16 **return** $(T_t[t], i \oplus T_e[t])$

proc $\mathcal{O}_c(i, k_w, s)$:
17 $h \leftarrow \perp$
18 **if** $i = H_1$ **then**
19 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
20 **else if** $i = H_2$ **then**
21 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
22 **return** h

proc $\mathcal{S}_c(w)$:
23 **if** $w \in W$ **then**
24 $k_w \leftarrow f(w)$
25 $s_c \leftarrow W[w]$
26 $c \leftarrow |s_c| - 1$
27 $r \leftarrow (k_w, s_c[c], c)$
28 $t_0 \leftarrow T[t]$
29 $i \leftarrow 0$
30 **while** $i < |t_0|$ **do**
31 $s \leftarrow s_c[i]$
32 $\widetilde{t}, \_ \leftarrow t_0[i]$
33 $T_{H_1}[(k_w, s)] \leftarrow T_t[\widetilde{t}]$
34 $T_{H_2}[(k_w, s)] \leftarrow T_e[\widetilde{t}]$
35 $i \leftarrow i+1$
36 **else**
37 $r \leftarrow \perp$
38 $t \leftarrow t+1$
39 **return** r

proc $\mathsf{SimH}_1(k_w, s)$:
40 **if** $(k_w, s) \notin T_{H_1}$ **then**
41 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
42 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$:
43 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$:
44 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
45 $W \leftarrow \emptyset$
46 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
47 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
48 **return** α

Alg. 32. The model of G_{14} client in pseudo-code. It removes the call to the random function from the update procedure. To prove this step, we used information flow techniques, as explained in Chapter 5.

$G_{14}.Client$:

$f : \{0,1\}^l \to \{0,1\}^l$ `// random function`
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha^\star$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \mathbb{N} \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \mathbb{N} \to \{0,1\}^{\lambda'}$
$t : \mathbb{N}$ `// timestamp`
$T : \{0,1\}^l \to (\mathbb{N} \times D_\alpha)^\star$

Client:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 **if** $w \notin W$ **then**
2 $s_c \leftarrow t_0 \leftarrow \emptyset$
3 $s \xleftarrow{\$} D_\alpha$
4 $c \leftarrow 0$
5 **else**
6 $t_0 \leftarrow T[t]$
7 $s_c \leftarrow W[w]$
8 $s \leftarrow \mathcal{B}_\tau(s)$
9 $c \leftarrow |s_c|$
10 $W[w] \leftarrow s_c || [s]$
11 $T[w] \leftarrow t_0 || [(t,i)]$
12 $T_t[t] \xleftarrow{\$} \{0,1\}^\lambda$
13 $T_e[t] \xleftarrow{\$} \{0,1\}^{\lambda'}$
14 $t \leftarrow t+1$
15 **return** $(T_t[t], i \oplus T_e[t])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
16 $h \leftarrow \perp$
17 **if** $i = H_1$ **then**
18 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
19 **else if** $i = H_2$ **then**
20 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
21 **return** h

proc $\mathcal{S}_c(w)$**:**
22 **if** $w \in W$ **then**
23 $k_w \leftarrow f(w)$
24 **if** f *has collisions* **then**
25 $b_c \leftarrow 1$
26 **return** $\perp$
27 $s_c \leftarrow W[w]$
28 $c \leftarrow |s_c| - 1$
29 $r \leftarrow (k_w, s_c[c], c)$
30 $t_0 \leftarrow T[t]$
31 $i \leftarrow 0$
32 **while** $i < |t_0|$ **do**
33 $s \leftarrow s_c[i]$
34 $\widetilde{t}, \_ \leftarrow t_0[i]$
35 $T_{H_1}[(k_w, s)] \leftarrow T_t[\widetilde{t}]$
36 $T_{H_2}[(k_w, s)] \leftarrow T_e[\widetilde{t}]$
37 $i \leftarrow i+1$
38 **else**
39 $r \leftarrow \perp$
40 $t \leftarrow t+1$
41 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
42 **if** $(k_w, s) \notin T_{H_1}$ **then**
43 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
44 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
45 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$**:**
46 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
47 $W \leftarrow \emptyset$
48 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
49 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
50 **return** α

Alg. 33. The model of G_{15} client in pseudo-code. It simulates results using the search pattern.

$\underline{G_{15}.Client}$:

$f : \{0,1\}^l \to \{0,1\}^l$ `// random function`
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \mathbb{N} \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \mathbb{N} \to \{0,1\}^{\lambda'}$
$t : \mathbb{N}$ `// timestamp`
$T : \{0,1\}^l \to (\mathbb{N} \times D_\alpha)^\star$

$\underline{\text{Client}}$:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 **if** $w \notin W$ **then**
2 $t_0 \leftarrow \emptyset$
3 $W[w] \xleftarrow{\$} D_\alpha$
4 $c \leftarrow 0$
5 **else**
6 $t_0 \leftarrow T[t]$
7 $s \leftarrow \mathcal{B}_\tau(W[w])$
8 $c \leftarrow |s_c|$
9 $T[w] \leftarrow t_0 || [(t, i)]$
10 $T_t[t] \xleftarrow{\$} \{0,1\}^\lambda$
11 $T_e[t] \xleftarrow{\$} \{0,1\}^{\lambda'}$
12 $t \leftarrow t + 1$
13 **return** $(T_t[t], i \oplus T_e[t])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
14 $h \leftarrow \perp$
15 **if** $i = H_1$ **then**
16 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
17 **else if** $i = H_2$ **then**
18 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
19 **return** h

proc $\mathcal{S}_c(w)$**:**
20 **if** $w \in W$ **then**
21 $k_w \leftarrow f(w)$
22 **if** f *has collisions* **then**
23 $b_c \leftarrow 1$
24 **return** $\perp$
25 $t_0 \leftarrow T[t]$
26 $c \leftarrow |t_0| - 1$
27 $i \leftarrow 0$
28 **while** $i < |t_0|$ **do**
29 **if** $i = 0$ **then**
30 $s \leftarrow W[0]$
31 **else**
32 $s \leftarrow \mathcal{B}_\tau(s)$
33 $\widetilde{t}, \_ \leftarrow t_0[i]$
34 $T_{H_1}[(k_w, s)] \leftarrow T_t[\widetilde{t}]$
35 $T_{H_2}[(k_w, s)] \leftarrow T_e[\widetilde{t}]$
36 $i \leftarrow i + 1$
37 $r \leftarrow (k_w, s, c)$
38 **else**
39 $r \leftarrow \perp$
40 $t \leftarrow t + 1$
41 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
42 **if** $(k_w, s) \notin T_{H_1}$ **then**
43 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
44 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
45 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$**:**
46 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
47 $W \leftarrow \emptyset$
48 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
49 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
50 **return** α

Alg. 34. The model of G_{16} client in pseudo-code. It removes the unused lines related to W from the update operation; at the same time it moves its filling to the search operation. This is the last step before the simulator.

$G_{16}.Client$:

$f : \{0,1\}^l \to \{0,1\}^l$ `// random function`
$\tau \in T$ `// trapdoor`
$W : \{0,1\}^l \to D_\alpha$
$T_{H_1} : \{0,1\}^l \times D_\alpha \to \{0,1\}^\lambda$
$T_t : \mathbb{N} \to \{0,1\}^\lambda$
$T_{H_2} : \{0,1\}^l \times D_\alpha \to \{0,1\}^{\lambda'}$
$T_e : \mathbb{N} \to \{0,1\}^{\lambda'}$
$t : \mathbb{N}$ `// timestamp`
$T : \{0,1\}^l \to (\mathbb{N} \times D_\alpha)^\star$

Client:

proc $\mathcal{U}_c(\mathsf{add}, w, i)$**:**
1 **if** $w \notin T$ **then**
2 $t_0 \leftarrow \emptyset$
3 **else**
4 $t_0 \leftarrow T[t]$
5 $T[w] \leftarrow t_0 || [(t, i)]$
6 $T_t[t] \xleftarrow{\$} \{0,1\}^\lambda$
7 $T_e[t] \xleftarrow{\$} \{0,1\}^{\lambda'}$
8 $t \leftarrow t + 1$
9 **return** $(T_t[t], i \oplus T_e[t])$

proc $\mathcal{O}_c(i, k_w, s)$**:**
10 $h \leftarrow \perp$
11 **if** $i = H_1$ **then**
12 $h \leftarrow (\mathsf{SimH}_1(k_w, s), \perp)$
13 **else if** $i = H_2$ **then**
14 $h \leftarrow (\perp, \mathsf{SimH}_2(k_w, s))$
15 **return** h

proc $\mathcal{S}_c(w)$**:**
16 **if** $w \in W$ **then**
17 $k_w \leftarrow f(w)$
18 **if** f *has collisions* **then**
19 $b_c \leftarrow 1$
20 **return** $\perp$
21 $t_0 \leftarrow T[t]$
22 $c \leftarrow |t_0| - 1$
23 $i \leftarrow 0$
24 **while** $i < |t_0|$ **do**
25 **if** $i = 0$ **then**
26 **if** $w \notin W$ **then**
27 $W[w] \xleftarrow{\$} D_\alpha$
28 $s \leftarrow W[0]$
29 **else**
30 $s \leftarrow \mathcal{B}_\tau(s)$
31 $\widetilde{t}, \_ \leftarrow t_0[i]$
32 $T_{H_1}[(k_w, s)] \leftarrow T_t[\widetilde{t}]$
33 $T_{H_2}[(k_w, s)] \leftarrow T_e[\widetilde{t}]$
34 $i \leftarrow i + 1$
35 $r \leftarrow (k_w, s, c)$
36 **else**
37 $r \leftarrow \perp$
38 $t \leftarrow t + 1$
39 **return** r

proc $\mathsf{SimH}_1(k_w, s)$**:**
40 **if** $(k_w, s) \notin T_{H_1}$ **then**
41 $T_{H_1}[(k_w, s)] \xleftarrow{\$} \{0,1\}^\lambda$
42 **return** $T_{H_1}[(k_w, s)]$

proc $\mathsf{SimH}_2(k_w, s)$**:**
43 omitted as analogous to SimH_1

proc $\mathcal{I}_c(1^\lambda)$**:**
44 $\alpha, \tau \leftarrow \mathcal{K}(1^\lambda)$
45 $W \leftarrow \emptyset$
46 $T_{H_1} \leftarrow T_t \leftarrow \emptyset$
47 $T_{H_2} \leftarrow T_e \leftarrow \emptyset$
48 **return** α