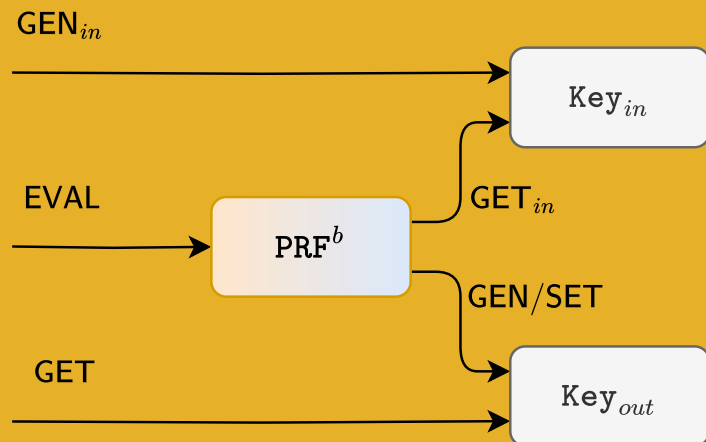


State-Separating Proofs and Their Applications

Konrad Kohbrok



State-Separating Proofs and Their Applications

Konrad Kohbrok

A doctoral thesis completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall C105 T2 of the school on 10 August 2023 at 14:00. Remote connection link: <https://aalto.zoom.us/j/2657129005>

Aalto University
School of Science
Department of Computer Science
Cryptography

Supervising professors

Prof. Chris Brzuska
Aalto University
Finland

Thesis advisors

Prof. Chris Brzuska
Aalto University
Finland

Antoine Delignat-Lavaud
Microsoft Research Cambridge
UK

Prof. Markulf Kohlweiss
University of Edinburgh
UK

Preliminary examiners

Prof. Tibor Jager
Bergische Universität Wuppertal
Germany

Prof. Dominique Unruh
University of Tartu
Estonia

Opponents

Prof. Tibor Jager
Bergische Universität Wuppertal
Germany

Aalto University publication series

DOCTORAL THESES 112/2023

© 2023 Konrad Kohbrok

ISBN 978-952-64-1355-6 (printed)

ISBN 978-952-64-1356-3 (pdf)

ISSN 1799-4934 (printed)

ISSN 1799-4942 (pdf)

<http://urn.fi/URN:ISBN:978-952-64-1356-3>

Unigrafia Oy
Helsinki 2023

Finland



Author

Konrad Kohbrok

Name of the doctoral thesis

State-Separating Proofs and Their Applications

Publisher School of Science**Unit** Department of Computer Science**Series** Aalto University publication series DOCTORAL THESES 112/2023**Field of research** Cryptography**Manuscript submitted** 3 April 2023**Date of the defence** 10 August 2023**Permission for public defence granted (date)** 12 June 2023**Language** English **Monograph** **Article thesis** **Essay thesis****Abstract**

Cryptographic protocols are commonly used to provide security for network traffic and digital interactions in general. Security means different things in different contexts. The most common security properties include confidentiality and authenticity of transmitted information, but cryptographic protocols can also provide more sophisticated security properties such as unlinkability or deniability. Cryptographers typically use *reduction proofs* to show that a given protocol indeed provides a certain type of security.

A prerequisite for such a proof is the formal description of the desired security notion. One way to achieve this formalization is the code-based game-playing framework introduced by Bellare and Rogaway [10], where security is encoded as a game played by a computationally bounded adversary. The game provides the adversary access to a set of oracles, which are defined through (pseudo-)code and which the adversary can query (or call) similar to functions in a computer program. Intuitively, the adversary wins the game if it can defeat the security of the protocol in question. A protocol is thus secure if the probability of an adversary winning the security game is sufficiently close to random chance. For example, we can define the security of a protocol aiming to allow two parties to agree on a secret key as follows: A protocol of this kind is secure if any adversary observing the protocol flow between two honest parties has a sufficiently low probability of distinguishing the resulting key from a randomly sampled string (thus expressing that the adversary has learned nothing about the real key).

Despite the structure and formalism introduced by Bellare and Rogaway's code-based game-playing framework, security definitions for larger protocols are still often complex and their proofs error-prone and hard to verify. A variety of frameworks exist to allow for composed definitions and modularization of proofs. With the *State Separating Proofs* framework, this thesis includes (in Publication I) one such framework. The SSP framework builds on that introduced by Bellare and Rogaway by leveraging the separation of state within a model to facilitate compositional proofs.

This thesis also includes several security proofs of real world protocols (or components thereof) to showcase the strengths and weaknesses of the SSP framework. In particular, the thesis includes a security proof of the key derivations of draft 11 of the Messaging Layer Security protocol (Publication III) the key schedule of the Transport Layer Security protocol (Version 1.3, Publication II), as well as a security protocol of a novel scheme for rotating signature keys (RSIG, Publication IV). This thesis also includes a computer-aided security proof of the *cryptobox* protocol (part of the NaCl library [11]) using the EasyCrypt [7] tool chain, where the SSP framework was used to guide both security modelling and proof (Publication V). Finally, we discuss the SSP framework in the context of related work such as other compositional frameworks and computer-aided cryptography.

Keywords Compositional Frameworks, Protocol Analysis, Key Exchange**ISBN (printed)** 978-952-64-1355-6**ISBN (pdf)** 978-952-64-1356-3**ISSN (printed)** 1799-4934**ISSN (pdf)** 1799-4942**Location of publisher** Helsinki**Location of printing** Helsinki **Year** 2023**Pages** 180**urn** <http://urn.fi/URN:ISBN:978-952-64-1356-3>

Preface

I would like to thank everyone who has supported me in writing this thesis. In particular, I would like to thank Chris Brzuska and Markulf Kohlweiss for their advice, for teaching me about cryptography and helping me grow along the way.

I am very grateful for the opportunity to meet and work with all the excellent researchers I have encountered during my PhD, chief among them my co-authors whom I would like to thank for the productive (and sometimes not so productive) discussions and the overall great time I have had while writing the papers in this thesis.

I would also like to thank Microsoft Research (MSR) for the financial support and for the invitations to the MSR labs in Cambridge and Redmond. I had a wonderful time during all of my visits and I am very thankful for the hospitality and the many things I learned from the everyone at MSR. This goes also for the F^{*} community, whose members taught me the way of type-based formal verification and patiently answered my numerous questions.

Outside of academia, I am grateful to my family and friends without the support of whom I could not have written this thesis.

Finally, I would like to thank the staff at Aalto University and the Hamburg University of Technology: Stephie for always being there for us PhD students and joining us in learning Finnish. Maisa and Johanna for the warm welcome to Finland and for helping me navigate the bureaucracy at Aalto.

Helsinki, June 28, 2023,

Konrad Kohbrok

Contents

Preface	1
Contents	3
List of Publications	5
Author's Contribution	7
1. Introduction: Taming complexity of cryptographic protocols	9
1.1 Security properties and goals	9
1.2 Security games	10
1.3 Code-based, game-playing definitions and proofs	10
1.4 Computer-aided cryptography	11
1.5 Contributions	12
2. State-Separating Proofs	15
2.1 BR-style games	15
2.2 Real vs. ideal distinguishing games	16
2.3 SSP-games vs. BR-games	16
2.4 Packaging and composing code for game-playing	17
2.5 SSP adversaries	17
2.6 Asserts	18
2.7 Modelling security in the SSP style	18
2.7.1 Modular security games	19
2.8 Basic proof patterns	21
2.8.1 Example: Double PRF	22
2.8.2 Proof.	23
2.8.3 On Reductions using SSP	27
2.9 Hybrid proofs and multi-instance games	28
2.9.1 Package-based hybrid arguments using SSP	28
2.9.2 Example: From Double-PRF to PRF-Chain	28
2.10 Development of the SSP framework	30
2.10.1 Origins	30

2.10.2	Key package styles	32
2.10.3	Use of asserts	33
3.	SSP proofs in practice	35
3.1	Provability as a protocol design goal	35
3.2	Handle construction and mapping	37
3.3	Models beyond the real-vs.-ideal paradigm	39
3.4	Summary and outlook	40
4.	Compositional Frameworks	41
4.1	Top-down frameworks	42
4.1.1	Simulation based security	42
4.1.2	Universal Composability	43
4.1.3	Abstract Cryptography	43
4.1.4	Constructive Cryptography	44
4.1.5	SSP and top-down frameworks	44
4.2	Bottom-up frameworks	44
4.2.1	“The Joy of Cryptography”	45
5.	The SSP framework and computer-aided cryptography	47
5.1	Relational F^*	47
5.2	SSProve: SSP formalized in Coq	48
5.3	Using SSP to guide EasyCrypt proofs	48
5.3.1	Packages in EasyCrypt	49
5.3.2	Flexibility in assert semantics	50
5.3.3	Package state	50
	Bibliography	53
	Publications	57

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

- I** Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, Markulf Kohlweiss. State Separation for Code-Based Game-Playing Proofs. In *Advances in Cryptology – ASIACRYPT 2018*, Pages 222-249, October 2018.
- II** Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, Markulf Kohlweiss. Key Schedule Security for the TLS 1.3 Standard. In *Advances in Cryptology – ASIACRYPT 2022*, Pages 621-650, December 2022.
- III** Chris Brzuska, Eric Cornelissen, Konrad Kohbrok. Security Analysis of the MLS Key Derivation. In *Proceedings of the IEEE Symposium on Security & Privacy 2022*, Pages 2535-2553, May 2022.
- IV** Cas Cremers, Britta Hale and Konrad Kohbrok. The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter. In *Proceedings of the 30th USENIX Security Symposium*, Pages 1847-1864, August 11-13 2021.
- V** François Dupressoir, Konrad Kohbrok, Sabine Oechsner. Bringing State-Separating Proofs to EasyCrypt - A Security Proof for Cryptobox. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, Pages 211-226, August 2022.

Author's Contribution

Publication I: “State Separation for Code-Based Game-Playing Proofs”

All authors of the article contributed equally.

Publication II: “Key Schedule Security for the TLS 1.3 Standard”

The author made major contributions to the overall design of the proof and participated in the writeup of the paper. All authors contributed to the writing and the proof development.

Publication III: “Security Analysis of the MLS Key Derivation”

The article is based on the Eric Cornelissen's master thesis. The author co-advised Eric together with Chris Brzuska. The core definitions, proof ideas and proof structure were developed by Eric, with refinements and feedback by the author and Chris. All authors contributed to the writing of the paper. The author specifically contributed to the comparison of the proven security guarantees with those of traditional key-exchange notions, as well as those used in related work. Finally, the author contributed to the paper's notion of key freshness.

Publication IV: “The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter”

All authors of the article contributed equally. The author worked in particular on the security model and wrote the proofs of Theorems 1 and 2 in the

appendix.

Publication V: “Bringing State-Separating Proofs to EasyCrypt - A Security Proof for Cryptobox”

The paper went through two rewrites and multiple proof approaches, where in total, each author had a roughly equal share of contributions, with each author contributing to all parts of the paper at some point. Broadly speaking Sabine Oechsner and the author mostly worked on the security model and on-paper proof, while François Dupressoir contributed the majority of the EasyCrypt proof and the discussion section.

1. Introduction: Taming complexity of cryptographic protocols

Cryptographic protocols such as the Transport Layer Security [38] (TLS) protocol facilitate the encryption of an estimated 85% of global internet traffic [29], and, as such, security protocols have become a vital part in the Internet's (security) architecture. The design and the intended *security properties* of these protocols thus need to be carefully considered.

1.1 Security properties and goals

Confidentiality and *authenticity* come to mind as basic security properties that users would expect from a protocol such as TLS: A secure channel between two parties should imply that both parties (or at least one in the case of TLS) know with whom they are communicating, and messages sent over said channel should be readable only by the intended recipient. However, venturing past these relatively simple security properties yields more complex constructs. For example, considering that one of the two parties involved get compromised: Should past communication still be protected despite the compromise? And should the compromised party be able to heal from a compromise eventually? How do we define these properties in case there are more than two parties involved in the protocol? At this point, defining security properties can become nuanced.

Once the designers of a given cryptographic protocol agree on a set of properties that their protocol shall fulfill, we can attempt to perform a *reduction proof*. A valid reduction proof states that a protocol fulfills its security properties if a certain set of *assumptions* holds. A typical assumption would be that a specific mathematical problem that the protocol is based on is hard to solve. Classic examples for mathematical problems that we can reliably create hard-to-solve instances for are the computation of the discrete logarithm in certain groups (i.e. given b and x , try to find a such that $b^a = x$), or finding solutions to noisy systems of linear equations. The application of this process is often referred to as *reduction-based security* or *provable security* (although the latter term can be criticized for

suggesting stronger guarantees than the reduction-based approach can actually provide, since security is only proven w.r.t. a set of assumptions and in a specific attack model rather than as an independent property).

Before we can reason about the security properties of a protocol mathematically, they first need to be formalized. This can often be done by defining a security property in terms of a *game*.

1.2 Security games

Goldwasser and Micali were the first to use *games* to formally define security [28]. Games typically provides a set of (potentially stateful and/or randomized) functions (called oracles) that an outsider can interact with by providing an oracle with an input and receiving an output in return. Game-based security then considers an algorithm called the *adversary* that interacts with the game and that is restricted in its computational power (e.g. only those adversaries are considered that run in a suitably bounded amount of time). A protocol is then considered secure, if we can upper bound the probability that any such adversary wins the game. For example, we can define confidentiality as a game, where the adversary can submit messages to an encryption oracle: In the beginning of the game, the oracle randomly samples a secret encryption key and flips a coin: If it is heads, it always encrypts the message m it receives from the adversary, else it encrypts the all-zero string of length $|m|$. The adversary wins the game if they can guess if the output of the encryption oracle is indeed the encrypted message or an encryption of zeroes. The intuition is that a protocol is secure if an adversary with a reasonable amount of (computational) resources cannot tell the difference between the actual ciphertext and a string that contains no information about the message save (inherently) its length (even if the message is chosen by the adversary).

Security proofs for game based definitions often use game-hops, for example as described by Shoup [41]. In this technique, the prover defines a sequence of games starting with the security definition and ending with a game that is unwinnable for the adversary, where with each “hop”, the game is changed only slightly. The prover can then show for each start and destination game of a hop via reduction to a given set of assumptions that they are indistinguishable to a computationally restricted adversary.

1.3 Code-based, game-playing definitions and proofs

Before the advent of definitional frameworks, cryptographers used prose to describe security games as we did in the previous example. However, complex security properties are hard to describe concisely in prose, and

rigorous reasoning about changes to individual details in the course of game-hops can be hard to capture for provers and hard to read and verify for readers.

The *code-based game-playing* framework by Bellare and Rogaway [10] refines the game-based approach by replacing prose with (*pseudo*-)code to define the games' behaviour, thus enabling more precise definitions.

The precision afforded by code-based definitions is particularly useful for complex security definitions and/or large protocols in the context of game-hopping proofs. This is because the changes from one game to the next are immediately visible in the code of the respective games, allowing both prover and reader to more easily follow the changes in game definitions for each hop, as well as the reasoning about the difficulty with which an adversary might be able to distinguish between the two games. We revisit code-based game-playing proofs in Section 2.1 and briefly summarize related frameworks for cryptographic proofs in Section 4.

1.4 Computer-aided cryptography

Going beyond frameworks as introduced by Bellare and Rogaway or structurally guided proof techniques as described by Shoup, Halevi, in his seminal 2005 paper [30], proposes another step towards reducing proof effort and improving proof readability and verifiability. In particular, he calls for a *computer-aided* approach to cryptographic proofs with the idea that a computer program can help a human prover in two ways: On the one hand, it can support the human prover by performing the tedious parts of a proof, leaving the creative challenge of finding a proof in the first place to the human. On the other hand, it can aid the human verifier, by automatically verifying said parts, leaving only the conceptually interesting parts of the proof (e.g. the security definitions and/or the bounds of the adversary) to review.

The two categories of tools for computer-aided cryptography that have since emerged roughly fall into two categories as already analyzed by Abadi and Rogaway [1]: Those reasoning in the *symbolic model* and those reasoning in the *computational model*.

Tools in the symbolic model category allow reasoning about abstractions of the individual elements of cryptographic protocols and their building blocks. For example, keys, ciphertexts, etc. are not seen as bitstrings, but rather as symbolic objects that can be used in the context of other objects or protocol functions, but not otherwise examined or manipulated. This approach allows powerful automatic reasoning even for complex protocols and security properties, see e.g. [23],[22] (Tamarin) and [13] (ProVerif) for comprehensive symbolic analysis' of the TLS protocol.

Tools using the computational model on the other hand follow the reduction-

based security approach, with a majority of them working in Bellare and Rogaway’s code-based game-playing framework. Examples include *CryptoVerif* [16] and *EasyCrypt* [7].

For a comprehensive overview over a number of tools of both categories, see Barbosa et al.’s systemization of knowledge paper on computer-aided cryptography for an overview [4].

1.5 Contributions

In this thesis, we present the SSP framework (Publication I), as well as a collection of works that use it to facilitate cryptographic proofs. The SSP framework is on the one hand inspired by computer-aided cryptography techniques and on the other hand a variant of Bellare and Rogaway’s code-based game-playing proofs framework that is not tied specifically to a computer-aided approach.

State-separating Proofs (SSP). The SSP framework can be seen as a variant of Bellare and Rogaway’s code-based game-playing framework [10] that allows the organization of code in a modular way. The partitioning of code based on the state it operates on allows for modular proofs that are easier to both write, as well as read and verify. In addition to the formal definition of the SSP framework (Publication I), we provide a brief introduction in Section 2 alongside a small example showcasing the advantages of the SSP framework.

Real-world examples. Besides the toy example in Section 2, this thesis also contains security proofs of real-world protocols (or parts thereof): In Publication II, we prove the security of the TLS 1.3 key schedule, Publication III contains a security proof of the cryptographic core of draft-11 of the Messaging Layer Security (MLS) protocol [5] and in Publication V we use the SSP framework to guide a security proof of the *cryptobox* protocol (part of the NaCl library [11]) using the *EasyCrypt* [7] tool chain. Finally, in Publication IV we introduce a simple signature key rotation protocol (RSIG), which we prove secure using the SSP framework. In Section 3, we discuss a few of the insights gained from the use of the SSP framework in these individual cases.

Related work. The SSP framework is neither the only nor the first framework that aims to facilitate modular proofs. On the contrary, a number of frameworks exist with similar goals, but differing approaches. In Section 4, we briefly introduce a few such frameworks that have been used in the recent past to analyse larger security protocols and discuss them in the context of the SSP framework.

Even though the SSP framework is geared towards use with pen-and-paper proofs, it has also been used to facilitate proofs using computer-aided

cryptography, Publication V being one of them. In Section 5, we briefly discuss two more such works. The first one, although incomplete, attempts modelling the SSP framework formally using relational logic. The second one is SSProve [2], a full formalization of the SSP framework using the Coq [42] proof assistant.

2. State-Separating Proofs

The State-separating proofs (SSP) framework structures and visualizes code-based game-playing proofs. SSPs were inspired by the modular code-based verification proof technique introduced by Fournet, Kohlweiss and Strub [26] and designed to ease security proofs of large and complex protocols such as TLS. The SSP framework can be understood as an evolution of the code-based game-playing proofs as introduced by Bellare and Rogaway (BR) [10].

This section provides an overview over the mechanics of SSPs. See Publication I for the complete formal definition and Section 2.10.1 for a brief discussion of its origins in code-based verification.

2.1 BR-style games

The SSP framework fundamentally builds on BR's approach of defining cryptographic security games using code as briefly introduced in Section 1. In particular, the code describes the functionality of the *oracles* the game makes available to the adversary. The code operates on the input the adversary provides when calling (or querying) an oracle, as well as the game's internal state. Oracles are thus similar to stateful functions or procedures in programming languages.

In BR's code-based game-playing approach, the state of a game is initialized by a specific initialization oracle which is run before the adversary first interacts with the game. The initialization procedure can also be used to provide the adversary with initial inputs.

Similarly, BR's games include a finalize procedure, which the adversary can call with its output (the nature of which depends on the way the game defines security). The finalize procedure can then perform additional computations to determine if the adversary has won.

2.2 Real vs. ideal distinguishing games

Micali, Goldreich and Wigderson [37] were the first who used a game-playing technique, where the adversary has to distinguish between two games: one exhibiting the real behaviour of a construction and the other its ideal behaviour. The ideal behaviour is one where the algorithms of the construction are used in the same way as in the real case, but their behaviour is changed such that it fulfills its functionality with perfect security. Here, perfect security means that an adversary interacting with the construction has no way of breaking its security.

For example, the goal of a pseudorandom function (PRF) is, given a random key k and an input x to return a pseudorandom byte string y of length $|x|$. The real behaviour of a given PRF construction would simply be to execute its *prf* algorithm with the given inputs. However, its ideal behaviour would be to always output a completely random byte string y .

Search-based games are an alternative to distinguishing games. Here, the adversary has to find and submit a value with one or more specific properties to win the game. While most SSP models, as well as all other compositional frameworks (cf. Section 4) rely on distinguishing games, it is also possible to model search-based games using SSP, which we will discuss in Section 3.3.

2.3 SSP-games vs. BR-games

Games in the SSP framework can be seen as variants of BR games with the difference that SSP games do not allow *initialize* or *finalize* procedures that are automatically executed in the beginning or the end of the game respectively.

Instead of an initialize procedure, SSP-style games rely on default-initialization of their state variables or require the adversary to call the oracles of a game in a specific order to ensure that the whole game state is initialized correctly. If the game needs to provide initial information to the adversary, an SSP-style game exposes an oracle that returns the relevant information or by initializing variables explicitly when they are first used. Jumping ahead, the adversary can then be made to call this initialization oracle first through the use of asserts (see Section 2.6).

Encoding finalize procedures in the context of a compositional framework such as SSP is not as straight-forward. As we discuss in more detail in Section 2.7, this is because a game can consist of multiple other games, each with its own finalize procedure. In that context, it is not clear when and how these procedures are called and what scope their effect has. We briefly discuss a potential way of implementing a finalize procedure using compositional techniques in Section 2.5.

2.4 Packaging and composing code for game-playing

The main feature of the SSP framework is that it allows us to divide code and state of a game into *packages*. Concretely, a package consists of a set of oracles (defined by code), as well as their shared state variables. We also say the package *provides* its oracles, as they can be queried (or called) by other packages. Packages are thus not dissimilar to objects in object oriented programming languages or functors in functional programming languages.

The set of oracles provided by a given package make up its *output interface*, while the *input interface* is the set of (external) oracles that are queried in turn by the package's own oracles. If a package P_1 queries an oracle of another package P_2 , we denote this by $P_1 \rightarrow P_2$ and say that the packages P_1 and P_2 are *composed*. Package composition will be covered in more detail in Section 2.7.1. If a package does not query oracles of other packages, we call that package a *game*, recovering the game notion introduced in Section 2.2.

2.5 SSP adversaries

In SSP an adversary against a particular tuple of games (G^0, G^1) with identical output interfaces is typically defined as a package providing a single oracle (RUN) that outputs their guess for the distinguishing bit $b \in \{0, 1\}$ and that has an input interface matching the output interface of $G^{0/1}$. We thus express the advantage of a given adversary \mathcal{A} in distinguishing two games G^0 and G^1 as the output of the following advantage function.

$$\epsilon_G(\mathcal{A}) := \left| \Pr [1 = \mathcal{A} \rightarrow G^0] - \Pr [1 = \mathcal{A} \rightarrow G^1] \right| \quad (2.1)$$

Alternatively, we sometimes use the following notation:

$$G^0 \stackrel{\epsilon_G(\mathcal{A})}{\approx} G^1. \quad (2.2)$$

As a compositional framework, the goal of SSP is simply to relate the security of a composed construction to that of its individual parts. It is thus suitable both for use with concrete security, as well as asymptotic security, where concrete security demands a fixed time bound and advantage ϵ (discussed, e.g., in Section 1.2 of [9]), while asymptotic security typically bounds security of all polynomial-time adversaries by a negligible function of the construction's security parameter (see e.g., Chapter 1.4.2. of [27]).

2.6 Asserts

In specific situations, it can be useful to restrict the behaviour of an adversary interacting with a game. In SSP games, this can be done through the use of *asserts* as shown in Figure 2.1, where an oracle asserts that a local variable has not previously been initialized.

```

GEN()
-----
assert  $k = \perp$ 
 $k \leftarrow \{0,1\}^\lambda$ 
...

```

Figure 2.1. Example for the use of an assert.

Each assert has a condition that depends on a (typically public or publicly computable) subset of the local package state. Generally, the assert is meant to ensure correct execution of the game, e.g. by preventing access of uninitialized variables or by forcing the adversary to run setup procedures prior to interacting with other oracles. For the sake of this introduction to the SSP framework, we say that our definitions only consider adversaries that don't violate any assert conditions. For a brief discussion of other assert semantics, see Section 2.10.3.

2.7 Modelling security in the SSP style

We now formalize the indistinguishability-based PRF security game outlined in Section 2.2 in the SSP style. We start by defining the real and the ideal games $\text{PRF}^{b,prf}$ in Figure 2.2 using pseudo-code, where the superscripts determine the game's *parameters* $b \in \{0,1\}$ is the *distinguishing bit* and *prf* is the actual PRF construction (although we will frequently elide the *prf* package parameter except when relevant). Parameterizing with $b \in \{0,1\}$ means that $\text{PRF}^{b,prf}$ defines two games: $\text{PRF}^{0,prf}$ and $\text{PRF}^{1,prf}$ and the goal of the adversary is to distinguish one from the other. For simplicity and to better contrast the two games, they are defined using the same code parameterized by b .

The game describes a multi-instance version of the traditional PRF security game, where the adversary can generate an arbitrary number of random symmetric keys and then interact with those keys through the use of administrative identifiers we call *handles*.

As noted in Section 2.3, there is no initialization function in SSP games, so the oracle has to include an **assert** condition at the start of the oracle enforces that the adversary can call EVAL only once.

We now divide the $\text{PRF}^{b,prf}$ games from Figure 2.2 into multiple packages such that we can later re-use it to define larger, composed security games.

```

EVAL( $h, x$ )
-----
if  $K[h] = \perp$ 
     $K[h] \leftarrow \{0, 1\}^\lambda$ 
if  $b$  then
    if  $Y[h, x] = \perp$ 
         $Y[h, x] \leftarrow \{0, 1\}^{|x|}$ 
     $y \leftarrow Y[h, x]$ 
else
     $y \leftarrow \text{prf}(k, x)$ 
return  $y$ 

```

Figure 2.2. Definition of the multi-instance $\text{PRF}^{b, \text{prf}}$ game, where λ is the key length of prf . The EVAL oracle takes as input a handle h and then either returns a randomly sampled value (if $b = 1$) or computes the value using the prf (if $b = 0$). The key for each input handle h is generated when h is first used as input. Similarly, outputs are cached, s.t. multiple calls using the same handle and input combination yield the same output even if $b = 1$.

2.7.1 Modular security games

On the surface, there is no reason why the games $\text{PRF}^{b, \text{prf}}$ shouldn't be defined as a the single package shown in Figure 2.2. However, modularization helps compose this game with others, as we will show in Section 2.8.

We now modularize $\text{PRF}^{b, \text{prf}}$ by introducing two additional packages: One to store the PRF's input key and one to store its outputs. Both packages need to provide oracles to facilitate the generation, storage and retrieval of values. Since these values are often keys, this package is called the $\text{Key}^{b, \lambda}$ package. The first parameter $b \in \{0, 1\}$ determines if the input keys are stored as they are or if the package instead stores a random key of the same length. The second parameter λ denotes the length of the stored keys. With λ representing the security parameter throughout this section, we say that it is a fixed, but arbitrary value. For convenience, we often elide the parameter λ and only write Key^b . See Figure 2.3 for the definition of the Key^b package. $\text{SET}(h, k)$ is used to either store the input key k (if $b = 0$) or a random key of length λ (if $b = 1$) under the handle h . If there is already a key stored under the handle h , it does nothing and returns. $\text{GET}(h)$ can in turn be used to retrieve a previously stored key. This kind of state-keeping package is at the heart of all works using the SSP framework so far, even though it differs slightly depending on the use-case (see Section 2.10.2).

The bit b is useful in multiple ways. On the one hand, it allows us to force all input keys to be randomly generated by using Key^1 for the PRF's input keys. On the other, we can use it to encode the PRF's ideal functionality: Instead of using the bit b of the PRF package to determine if keys are generated randomly, we can instead use the bit b of the output Key package instance. Jumping ahead slightly, the latter enables the graph-based

$\text{SET}(h, k)$	$\text{GET}(h)$
if $K[h] \neq \perp$	assert $K[h] \neq \perp$
return	return $K[h]$
if $b = 1$ then	
$k \leftarrow \{0, 1\}^\lambda$	
$K[h] \leftarrow k$	

Figure 2.3. Definition of the oracles of the Key package.

game-hopping technique described in Section 2.8.

The modularization steps described above preserve the original functionality of the game with one exception: An adversary interacting with the game has to generate the input key explicitly by calling SET before they can call EVAL and collect the output afterwards (by calling GET) rather than it being returned by EVAL. See Figure 2.4 for the definition of the PRF package after modularization.

$\text{EVAL}(h, x)$
$k \leftarrow \text{GET}(h)$
$y \leftarrow \text{prf}(k, x)$
$h_{out} \leftarrow (h, x)$
$\text{SET}(h_{out}, y)$

Figure 2.4. Definition of the PRF^{prf} package. The definition of h_{out} ensures that fresh output values are only generated once per (handle, input)-combination.

We will later see that the first step allows other games to provide the key for the PRF and the second allows other games in turn to use the output of the PRF as their own input key.

The SSP framework uses graphs to define and visualize modular packages. Figure 2.5a shows the game before modularization and Figure 2.5b after. We use orange and blue to highlight packages with a distinguishing bit, where $b = 0$ and $b = 1$ respectively.

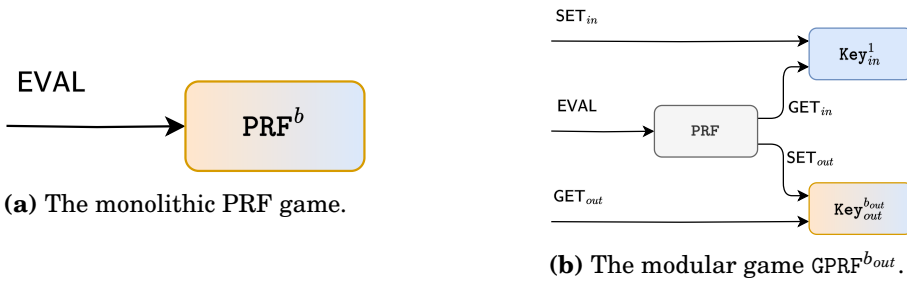


Figure 2.5. The monolithic and the modular variant of the PRF game.

Since the composed game includes two instances of the Key package, we have to distinguish the two: We use the subscript in for the instance of the

Key package holding the input keys and out for the one holding the outputs. Similarly, we will use these subscripts to disambiguate the names of the oracles provided by these packages. In fact this type of disambiguation is a formal requirement of the SSP framework in cases where a (potentially composed) package would otherwise provide two oracles of the same name. Generally, we use the term *package* to denote the definition of the package and *package instance* to denote a particular instance of a given package. For example, Key_{in} is an instance of the Key package. We will either be explicit about the difference between definition and instance where it matters, or, if multiple instances of a given package exists (as in this case with the two Key package instances) we will use labels when talking about a specific package instance. For convenience, if only one instance of a given package exists we use the name of the package to denote both the package instance and the package definition. The distinction between packages and package instances was not clear initially (in Publication I) and has been solved in different ways in different works using SSP. We will revisit this issue briefly in Section 5.3.1.

See Figure 2.5b for the definition of the fully modularized game $GPRF^{b_{out}}$, where the prefix G is used to indicate that this package is indeed a game, i.e. a package that does not make calls to oracles provided by other packages. Finally, our security statement for $GPRF^{b_{out}}$ security can be expressed as shown in Figure 2.6.

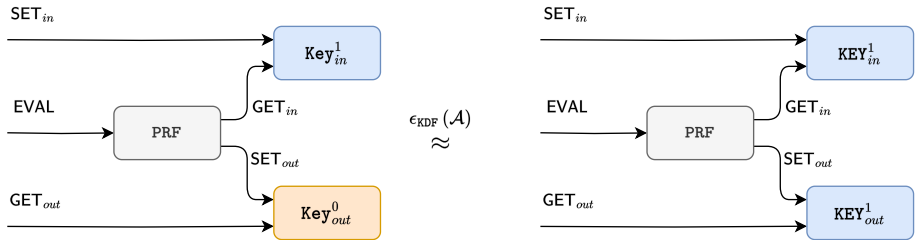


Figure 2.6. PRF security: For all adversaries \mathcal{A} against $GPRF^{b_{out}}$ security, let $\epsilon_{GPRF}(\mathcal{A})$ be the function that outputs the advantage of the adversary \mathcal{A} in distinguishing between the game instances $GPRF^0$ and $GPRF^1$.

2.8 Basic proof patterns

We now introduce basic proof patterns that are enabled by SSP's package algebra and its properties and that are used extensively, especially in the key schedule and key derivation security proofs in Publication II and Publication III, but also in the computer-aided proof of the Cryptobox protocol in Publication V.

2.8.1 Example: Double PRF

We use two instances of the modularized PRF definition from Section 2.7.1 to conduct a security proof for a simple two-step key derivation function (KDF)-chain, a simplified version of the KDF-chains used in key schedule design in general and, for example, in the Double-Ratchet algorithm of the Signal specification [33] in particular.

More concretely, our construction (see Figure 2.7) takes as input a key k_0 , as well as another input x_0 , performs a *prf* computation and uses the result (k_1) together with another input x_1 to perform another *prf* computation, yielding k_2 . We say that the construction is secure if k_2 is indistinguishable from a random string of the same length as long as the input key k_0 is sampled uniformly at random and not directly accessible to the adversary.

```

    dprf( $k_0, x_0, x_1$ )
     $k_1 \leftarrow \text{prf}(k_0, x_0)$ 
     $k_2 \leftarrow \text{prf}(k_1, x_1)$ 
    return  $k_2$ 
  
```

Figure 2.7. Definition of the Double-PRF construction.

We define the corresponding security notion in a modular fashion in Figure 2.9. The DPRF package (see Figure 2.8) exposes a single oracle DOUBLE-EVAL that takes as input a handle h , as well as the values x_0 and x_1 , fetches the key k_0 from Key_{in}^1 using h , performs the computation described above and finally stores the resulting value k_2 in $\text{Key}_{out}^{b_{dprf}}$ via the SET oracle using the handle h_{out} constructed from h, x_0 and x_1 .

```

    DOUBLE-EVAL( $h, x_0, x_1$ )
     $k_0 \leftarrow \text{GET}(h)$ 
     $k_2 \leftarrow \text{dprf}(x_0, x_1)$ 
     $h_{out} \leftarrow (h, x_0, x_1)$ 
    SETout( $h_{out}, k_2$ )
  
```

Figure 2.8. Definition of the DPRF package.

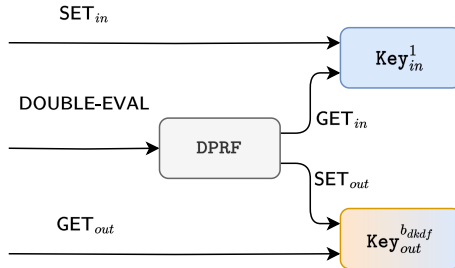


Figure 2.9. Composed high level security notion $\text{GDPRF}^{b_{dprf}}$.

2.8.2 Proof.

We can now conduct a reduction proof in the game hopping style that consists of the following steps. See Figure 2.11 for a visual proof overview.

1. We build a modular reduction game $\text{GMod-DPRF}^{b_{mid}, b_{out}}$ that consists of a reduction package Mod-DPRF composed with two composed $\text{GPRF}^{b_{out}}$ instances (see Figure 2.10).
2. We perform a number of game hops that relate the $\text{GMod-DPRF}^{0,0}$ to $\text{GMod-DPRF}^{1,1}$ (see Figures 2.11b and 2.11c).
3. We establish that $\text{GMod-DPRF}^{b,b}$ and GDPRF^b are functionally equivalent for $b \in \{0,1\}$, i.e. that the oracles they provide have the same behaviour (see Figures 2.11a and 2.11d).
4. We conclude by calculating the upper bound of the adversarial advantage in distinguishing $\text{GMod-DPRF}^{0,0}$ and $\text{GMod-DPRF}^{1,1}$ and thus (due to the result of step 2) GDPRF^0 and GDPRF^1 . This upper bound is exactly the sum over the advantage in distinguishing the games of each hop of step 2.

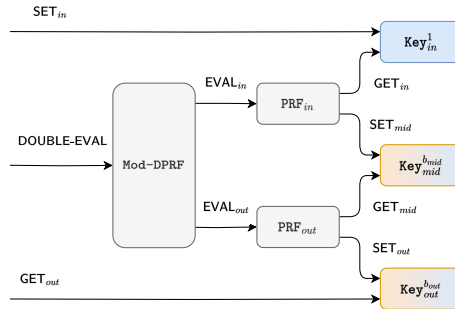
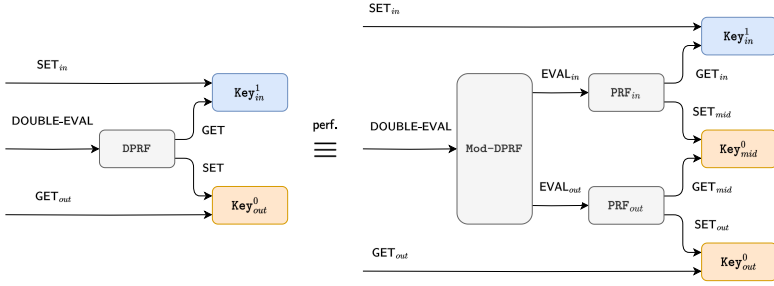
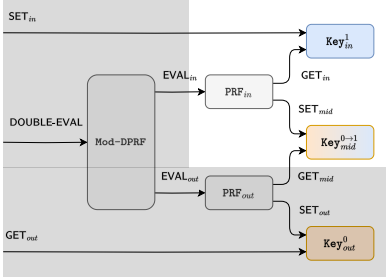


Figure 2.10. The composed game $\text{GMod-DPRF}^{b_{mid}, b_{out}}$.

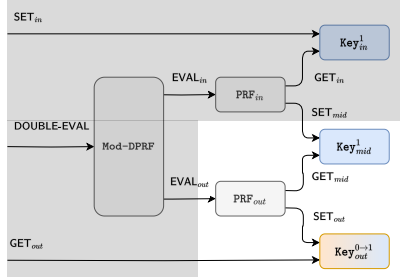
Step 1: A reduction package. Before we can define our reduction packages, we first construct a modularized version of our security notion $\text{GDPRF}^{b_{dprf}}$, which we call GMod-DPRF . We construct it as follows: First, we introduce a new package Mod-DPRF has the same output interface as $\text{GDPRF}^{b_{dprf}}$. The Double-PRF construction (Figure 2.7) has two calls to a PRF, each with its own key. We represent each PRF instance using one partial instance of GPRF and call the first one GPRF_{in} and the second one GPRF_{out} . The instances are partial in that they share a Key package instance: Since the output of the first PRF call is the input of the second, we re-use the output Key package instance of the first GPRF as the input Key package of the second one and call it $\text{Key}_{mid}^{b_{mid}}$. We then compose all of the packages



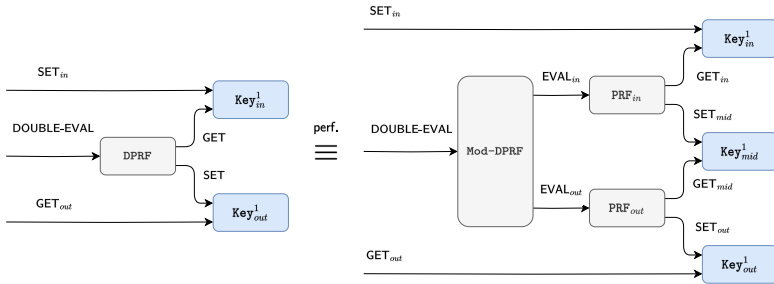
(a) Equivalence between GPRF^0 and composed reduction game $\text{GMod-DPRF}^{0,0}$



(b) Game hop 1. The gray area defines the package $\mathcal{R}_{\text{GPRF}_{in}}$ with an instance of GPRF in the top right corner.



(c) Game hop 2. The gray area defines the package $\mathcal{R}_{\text{GPRF}_{out}}$ with an instance of GPRF in the bottom right corner.



(d) Equivalence between GPRF^1 and composed reduction game $\text{GMod-DPRF}^{1,1}$

Figure 2.11. Overview over the Double-PRF security proof steps.

described above, defining Mod-DPRF 's DOUBLE-EVAL oracle such that it first calls EVAL of GPRF_{in} , followed by a call to the EVAL oracle of GPRF_{out} (see Figure 2.12). See Figure 2.10 for the resulting game $\text{GMod-DPRF}^{b_{mid}, b_{out}}$.

Step 2: Game hopping. Next, we perform a series of game-hops, where we idealize the two PRF building blocks. The style of game-hopping shown in this section highlights one of the main benefits of SSP: The re-usability of code due to packaging. Together with the associativity of package composition (which follows from the isolation of state) our reduction algorithms

$$\begin{array}{c}
\text{DOUBLE-EVAL}(h, x_0, x_1) \\
\hline
\text{GPRF}_{in}.\text{EVAL}(h, x_0) \\
h' \leftarrow (h, x_0) \\
\text{GPRF}_{out}.\text{EVAL}(h', x_1)
\end{array}$$

Figure 2.12. Definition of the DOUBLE-EVAL oracle of the Mod-DPRF package. We prepend the name of the providing package to the oracle name to disambiguate the EVAL oracles.

will follow immediately from the construction in question. See Figure 2.11 for a visual overview over the sub-steps that make up Step 2.

Looking at Figure 2.11b, we can see that the graph is identical to the preceding Figure 2.10, except that we have isolated GPRF_{in}^0 (an instance of GPRF^0 as defined in Figure 2.5b) in the top-right corner, that includes the Key_{mid} package. Additionally, we have marked the remaining composed package in gray, which we call the composed reduction package $\mathcal{R}_{\text{GPRF}_{in}}$. Since package composition is associative (see Lemma 6 in Section 2.1 of Publication I), the following holds

$$\mathcal{A} \rightarrow (\mathcal{R}_{\text{GPRF}_{in}} \rightarrow \text{GPRF}_{in}^0) = (\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}}) \rightarrow \text{GPRF}_{in}^0$$

Since the composed package $\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}}$ matches our definition of an adversary, it follows from our security assumption (PRF security, Figure 2.6) that we can replace the GPRF_{in}^0 with GPRF_{in}^1 incurring an adversarial advantage of $\epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}})$. This in turn gives us

$$\text{GMod-DPRF}^{0,0} \stackrel{\epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}})}{\approx} \text{GMod-DPRF}^{1,0}, \quad (2.3)$$

hop from Figure 2.11b to Figure 2.11c.

This first idealization step, resulted in bit b_{mid} turning from 0 to 1, which allows us to perform the second idealization step.

The game hop from Figure 2.11c to Figure 2.11d follows the same pattern: We isolate GPRF_{out}^0 (another instance of GPRF^0) in the lower right corner of $\text{GMod-DPRF}^{1,0}$, with the composed reduction package $\mathcal{R}_{\text{GPRF}_{out}}$ marked in gray. Using our PRF security assumption (Figure 2.6) again, we can idealize the second part of the double PRF, which gives us

$$\text{GMod-DPRF}^{1,0} \stackrel{\epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}})}{\approx} \text{GMod-DPRF}^{1,1}. \quad (2.4)$$

Using Eq. 2.1 together with the fact that $\mathcal{R}_{\text{GPRF}_{out}} \rightarrow \text{GPRF}^1 = \mathcal{R}_{\text{GPRF}_{in}} \rightarrow$

GPRF⁰ (see Figures 2.11b and 2.11c), we can now compute $\epsilon_{\text{GMod-DPRF}}(\mathcal{A})$:

$$\begin{aligned}
 & \epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}}) + \epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}}) \\
 = & \left| \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}} \rightarrow \text{GPRF}^0] - \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}} \rightarrow \text{GPRF}^1] \right| \\
 & + \left| \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}} \rightarrow \text{GPRF}^0] - \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}} \rightarrow \text{GPRF}^1] \right| \\
 \geq & \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}} \rightarrow \text{GPRF}^0] - \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}} \rightarrow \text{GPRF}^1] \\
 & + \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}} \rightarrow \text{GPRF}^0] - \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}} \rightarrow \text{GPRF}^1] \\
 = & \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}} \rightarrow \text{GPRF}^0] - \Pr[1 = \mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}} \rightarrow \text{GPRF}^1] \\
 = & \epsilon_{\text{GMod-DPRF}}(\mathcal{A})
 \end{aligned}$$

Which in particular yields

$$\epsilon_{\text{GMod-DPRF}}(\mathcal{A}) \leq \epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}}) + \epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}}). \quad (2.5)$$

Step 3: Proving functional equivalence. In this step, we have to prove that the two games in question are functionally equivalent. Generally, for two games G_1 and G_2 to be functionally equivalent, they have to consider the same class of adversaries (which means they have the same assert restrictions) and for all such adversaries \mathcal{A} , we have

$$|\Pr[1 = \mathcal{A} \rightarrow G_1] - \Pr[1 = \mathcal{A} \rightarrow G_2]| = 0.$$

Equivalence of two games is hard to prove (and verify) on paper. Two techniques for such proofs are inlining of oracle code (in the case of composed games), followed by a side-by-side comparison of the result, as well as proving invariants over package state across oracle calls. Both can work well depending on the individual games to compare, but remain hard to verify manually. However, a much more elegant way is to leave these the code equivalence steps to mechanized proof assistants. We discuss a few instances of this approach in Section 5. For brevity, here we only discuss the code equivalence proof steps in this example conceptually and elide a detailed inlining proof.

The first step in our equivalence proof is to prove that the construction $\text{GMod-DPRF}^{b_{mid}, b_{out}}$ (Figure 2.10) is indeed equivalent to our original security games GDPRF_{dprf}^b (Figure 2.9) and thus correctly simulates the original security games to the adversary. For the purpose of this proof, we have to prove equivalence for both the real ($b_{dprf} = b_{mid} = b_{out} = 0$) and the ideal case ($b_{dprf} = b_{mid} = b_{out} = 1$).

In the real case, the equivalence follows from the construction of $\text{GMod-DPRF}^{b_{mid}, b_{out}}$. The DOUBLE-EVAL oracle of both GDPRF^0 and $\text{GMod-DPRF}^{0,0}$ calls the PRF twice, using the input of the first call to key the second. The only difference being that $\text{GMod-DPRF}^{0,0}$ stores the intermediate key (k_1) in a separate Key package instance, before immediately retrieving it again for the next EVAL call (Figure 2.12).

In the ideal case, the oracles differ in their definition, but the output distribution remains the same. In particular, since we have $b_{mid} = 1$, the second call to EVAL in $\text{GMod-DPRF}^{1,1}$'s DOUBLE-EVAL (Figure 2.12) doesn't use the actual result of the first PRF call as input for the second. Instead, it uses the random value that is sampled when storing the result of the first call in Key_{mid}^1 via the SET oracle. However, even the output of the second PRF computation is ignored by the Key_{out}^1 package in favor of a randomly sampled value. Since this behaviour matches that of GDPRF^1 's DOUBLE-EVAL (Figure 2.9), both oracles are functionally equivalent.

Finally, $\text{GMod-DPRF}^{0,0} = \text{GDPRF}^0$ and $\text{GMod-DPRF}^{1,1} = \text{GDPRF}^1$ yields

$$\text{GMod-DPRF}^{b,b} = \text{GDPRF}^b \quad (2.6)$$

for $b \in \{0, 1\}$.

Note that for both GDPRF^1 and $\text{GMod-DPRF}^{1,1}$, the call to DOUBLE-EVAL could be replaced by a simple call to SET of Key_{out}^1 , which randomly samples the result visible to the adversary. The result is thus independent of the rest of the (composed) package state and essentially captures the ideal functionality of the game.

Step 4: Adversarial advantage. We can now conclude by combining Eq. 2.6 from Step 3 with Eq. 2.5 from Step 2:

$$\text{GDPRF}^0 \stackrel{\epsilon_{\text{GMod-DPRF}}(\mathcal{A})}{\approx} \text{GDPRF}^1, \quad (2.7)$$

where

$$\epsilon_{\text{GDPRF}}(\mathcal{A}) = \epsilon_{\text{GMod-DPRF}}(\mathcal{A}) \leq \epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{in}}) + \epsilon_{\text{GPRF}}(\mathcal{A} \rightarrow \mathcal{R}_{\text{GPRF}_{out}}),$$

and with the packages $\mathcal{R}_{\text{GPRF}_{in}}$ and $\mathcal{R}_{\text{GPRF}_{out}}$ as defined in Figures 2.11b and 2.11c respectively.

2.8.3 On Reductions using SSP

The small example proof detailed above highlights the two main advantages of SSP: The ability to re-use code in the construction of reductions and the ease with which simulation correctness can be shown for such reductions based on graphs.

Both advantages become visible in Figures 2.11b and 2.11c. Here, the composed reduction packages $\mathcal{R}_{\text{GPRF}_{in}}$ and $\mathcal{R}_{\text{GPRF}_{out}}$ are trivially constructed from the sub-graph highlighted in gray, thus eliminating the need to define a dedicated reduction package.

The same graphs also allow the reader to verify that (by construction of the respective reduction) the game correctly simulates the original security game to the adversary, as the graph defining the reduction composed with its target security notion is exactly the same as that defining the original game.

2.9 Hybrid proofs and multi-instance games

In this section we showcase how the SSP framework can be used to easily conduct a hybrid argument by extending the GDPRF example from the previous section to a KDF chain. We first briefly summarize the hybrid argument recipe detailed in Appendix B of Publication I and then continue the example.

2.9.1 Package-based hybrid arguments using SSP

The hybrid argument recipe described in Appendix B of Publication I is a standard hybrid argument framed in terms of SSP packages. The recipe relates the advantage of an adversary in distinguishing a pair of multi-instance games Multi^b , $b \in \{0, 1\}$ to that of another adversary distinguishing a pair of single instance games Game^b , $b \in \{0, 1\}$ using a reduction package \mathcal{R} with the following advantage:

$$\epsilon_{\text{Multi}}(\mathcal{A}) \leq n \cdot \epsilon_{\text{Game}}(\mathcal{A} \rightarrow \mathcal{R}) \quad (2.8)$$

To instantiate the recipe, we require a hybrid package H^i and a reduction package \mathcal{R}^i with $i \in \mathbb{N}$. In particular, the extremes of the hybrid have to be perfectly equivalent to the Multi^b games:

$$\text{Multi}^0 \equiv H^0 \quad (2.9)$$

$$\text{Multi}^1 \equiv H^n \quad (2.10)$$

Similarly, the reduction composed with the single-instance games have to be equivalent to the hybrids:

$$\mathcal{R}^i \rightarrow \text{Game}^0 \equiv H^i \quad (2.11)$$

$$\mathcal{R}^i \rightarrow \text{Game}^1 \equiv H^{i+1} \quad (2.12)$$

If the two conditions above hold, the recipe yields Eq. 2.8, where \mathcal{R} is defined as the package that samples i uniformly at random and calls \mathcal{R}_i .

We prove that Eq. 2.8 holds by summing up the adversarial advantages in distinguishing subsequent package pairs from H^0, H^1 to H^{n-1}, H^n . For more details and the full proof, as well as another example application, see Appendix B of Publication I.

2.9.2 Example: From Double-PRF to PRF-Chain

We now revisit the GDPRF example from Section 2.8 and extend it into a chain. In particular, we consider a construction, where the double PRF iteration can be chained n , where $n \in \mathbb{N}$ is a constant. With each chain, the output key of the previous iteration is used as the input key of the next. This construction is a slightly simplified version of the PRF chain used

in the Double-Ratchet algorithm, which in turn constitutes a core part of the Signal protocol [33]. We provide a modular SSP definition of such a construction in Figure 2.13.

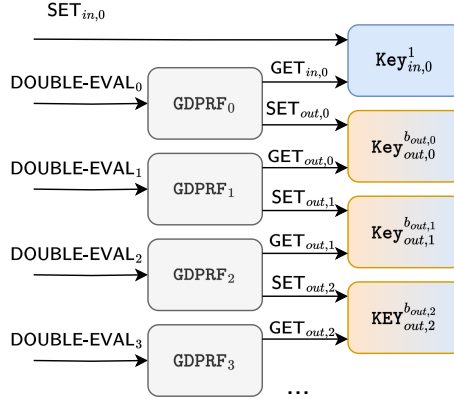


Figure 2.13. The composed game $\text{GDPRF-Chain}^{b_1, \dots, b_n}$ where $\forall i : b_i \in \{0, 1\}$.

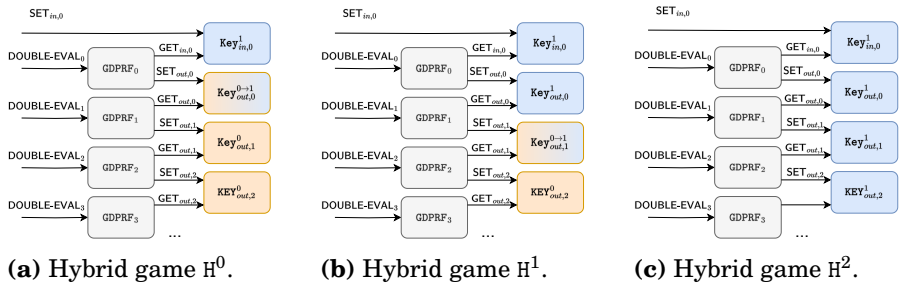


Figure 2.14. Steps through the hybrid proof of $\text{GDPRF-Chain}^{b_1, \dots, b_n}$.

Let $\text{GDPRF-Chain}^{b_1, \dots, b_n}$ be the package defined in Figure 2.13. We can now instantiate the hybrid argument recipe with $\text{Multi}^b := \text{GDPRF-Chain}^{b, \dots, b}$, and $\text{Game}^b := \text{GDPRF}^b$ for $b \in \{0, 1\}$. We define the hybrid H^i for $0 \leq i \leq n$ to be equal to GDPRF-Chain with the idealization bits $b_1 \dots b_i = 1$ and $b_{i+1} \dots b_n = 0$, such that we have $H^0 = \text{GDPRF-Chain}^{0 \dots 0}$ and $H^1 = \text{GDPRF-Chain}^{1 \dots 1}$. Finally, we define our reduction \mathcal{R}_i as shown in Figure 2.15.

Since \mathcal{R}_i is defined such that the i th GPRF instance in the chain is replaced by the composed instance of Game , all that remains is to instantiate the hybrid argument recipe as described in Section 2.9.1, which gives us

$$\epsilon_{\text{GDPRF-Chain}} \leq n \cdot \epsilon_{\text{GDPRF}}(\mathcal{A} \rightarrow \mathcal{R}),$$

where n is the length of the chain and \mathcal{R} defined as in the recipe. This concludes our hybrid proof.

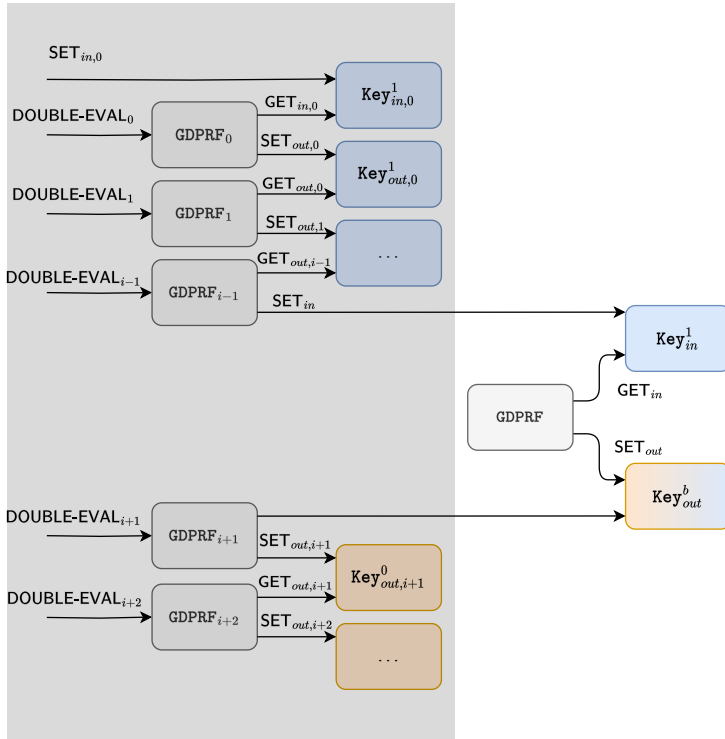


Figure 2.15. Definition of the reduction \mathcal{R}_i (gray) composed with GDPRF^b .

2.10 Development of the SSP framework

In this section we briefly discuss the origins of the SSP Framework (Section 2.10.1), as well as the changes to the SSP framework over time.

2.10.1 Origins

The original motivation behind the SSP framework was to facilitate the communication about modular proofs of large protocols. It was in particular inspired by the code-based verification technique introduced by Fournet, Kohlweiss and Strub (FKS) [26] and its use in the various works that make up the miTLS project such as [14], [25], [15] and others¹.

*Code-based verification using F^**

Different works that based on code-based verification as introduced by FKS have used different verification toolchains to implement it. Here, we focus on the technique as implemented by Bhargavan, Delignat-Lavaud, Fournet, Kohlweiss, Pan, Protzenko, Rastogi, Swamy, Zanella-Béguelin and Zinzindohoué [25], who use F^* , a functional programming language that allows the expression and verification of properties of a given piece of

¹See <https://www.mitls.org> for an overview.

code.

The modular, code-based verification technique by FKS fundamentally relies on packaging code and state in a similar way as the SSP framework. The authors of [25] make use of F^* 's *module* system, where each module describes both the real and the ideal functionality of a cryptographic primitive or protocol. An F^* module is thus comparable to a game in SSP-terms, although when using the SSP framework real and ideal behaviours are described in two individual games.

Similar to SSP packages, F^* modules can have an *interface*, which details the types of each of the module's functions. F^* 's refinement types allow the encoding of the properties provided by each function depending on its inputs.

The interface can then be used in the context of a reduction proof in two ways: either as an assumption in a reduction proof, or as the target security notion. In either case, the proof pattern is (perhaps unsurprisingly) similar to the one shown in Section 2.8: Given a target security notion, we implement the protocol using the real-or-ideal interfaces of the building blocks such that the protocol exhibits its real or ideal behaviour respectively if all of its building blocks do.

Given such a definitional approach, F^* 's type-checker can ensure that the game hop sequence is correct (cf. Step 2 in Section 2.8.2) and that the equivalence proofs hold (cf. Step 3 in Section 2.8.2). Correctness of the game hop sequence is ensured by the requirements in the interfaces of the individual building blocks. This is again similar to the pattern described in Section 2.8, where we could only idealize the PRF after ensuring that the keys it uses are indeed randomly sampled. The PRF would thus require in its interface that ideal behaviour can only be shown if the keys used are indeed sampled randomly. In turn, the type-checker verifies that the implementation correctly implements the interface, thus proving equivalence of the protocol (the semantics of which are described in the interface) and its implementation using the individual building blocks.

To summarize, if we successfully implement the interface of the target security notion (which is confirmed by the F^* typechecker), we have shown that the building blocks can be idealized in turn and that the protocol shows ideal behaviour if its building blocks do.

In addition to the security proofs, FKS' code-based verification enables its users to provide a direct link between the proven security (through the module interfaces) and the executable code extracted from the underlying programming language. With pen-and-paper proofs (based on the SSP framework or otherwise), this connection is hard to establish and verify.

One disadvantage of FKS' technique is that it requires that the summation of adversarial advantages incurred by idealizing the modularly implemented game is to be done manually and stated on paper rather than as part of the code.

On the other hand, modelling cryptographic games in F^* allows the prover to encode asserts (as discussed in Section 2.6) using F^* 's refinement types. Similarly, the reader can skip implementation details and review type-level specifications (which are checked by F^* 's type-checker) only. In particular, the type-checker can automate equivalence proofs (such as those discussed in Section 2.8.2) that are tedious to write and verify on paper.

In summary, FKS' code-based verification technique is a great way of linking computational security proofs of large protocols to executable code, even though a small part still remains to be done on-paper. While the SSP framework is primarily aimed at pen-and-paper proofs, its application to computer aided cryptography (see Section 5) has already yielded work that allows a fully machine-checked proof without reliance on pen-and-paper arguments.

2.10.2 Key package styles

Different proofs written using the SSP framework have used different definitions for the Key package. Publication I (being the earliest work using the SSP framework) uses a very basic Key package that resembles the one introduced in Section 2.7.1, although it does not include the distinguishing bit b . Instead, it provides two distinct oracles GEN and SET, where the former always generates a random key and the latter stores the input key, leaving it to the adversary or composed package to determine which oracle to call (See Definition 16 in Publication I). The game-hopping style exemplified in Section 2.8 is then enabled by examining the interfaces: Recalling the example in the aforementioned section, the second GPRF instance can only be idealized if its input Key package does not expose the SET oracle. The latter is the case exactly if the first GPRF instance is idealized, since an idealized GPRF instance (i.e. $GPRF^1$) does not query the SET oracle.

In their analysis of the NPRF primitive [19], Brzuska and Winkelmann first move the distinguishing bit to the Key package, resulting in the style of Key package defined in Figure 2.3. This new style of Key package enables the game-hopping style from Section 2.8 based on package instances rather than interfaces.

The same style is also used in subsequent works such as Publication II and Publication III, although with minor changes, e.g. to encode additional guarantees such as collision resistance or to allow for storage of dishonest keys (where SET always stores the input key independent of the bit b). Notably, in Publication II, the newer style of Key package also allows for a simulation-based argument in the proof, since the Key package fully encapsulates the ideal functionality of the key schedule. For a more in-depth discussion of this proof technique, see Section 5 of Publication II.

2.10.3 Use of asserts

Restricting the behaviour of adversaries is not specific to the SSP framework and so different methods of restriction are used depending on the needs of the security model. Even different works using the SSP framework are using different assert semantics.

Publication I defines the assert semantic such that the security game in question only considers the class of adversaries that never violate asserts. In contrast, Publication II and Publication III define assert to make the oracle in which it is used to return a special symbol to the adversary if it is violated.

We elide an in-depth discussion on adversary restriction in cryptographic model in general here and instead refer the reader to Section 1 of [8]. A more recent method of adversary control worth highlighting, however, was introduced by Rogaway and Zhang and is called *oracle silencing* [39]. In the silencing semantics, violation of an assert causes the oracle to return a special symbol directly to the adversary. After the initial assert violation, all other oracles also immediately return the same special symbol without further execution, thus preventing the adversary from further interacting with the game, while still allowing them to terminate and return a value. In the context of the SSP framework, this method raises implementation questions, because it relies on somewhat global state: To either trigger the silence, or determine if they should exhibit silenced behaviour, all oracles of a (composed) game have to access a flag that indicates if a silencing event has occurred. Thus, implementing an assert with this semantic technically requires an additional SILENCING package as defined in Figure 2.16.

$\frac{\text{SET}()}{s \leftarrow 1}$	$\frac{\text{GET}()}{\mathbf{return } s}$
---------------------------------------	---

Figure 2.16. Definition of the SILENCING package.

Every oracle would then call GET before execution and return the special symbol if the returned value is 1. Triggering an assert condition would consequently lead to a call to SET, thus locking the adversary out.

While the SILENCING package has not yet been used in an SSP-style proof, its use should be possible even in a compositional setting, as both SET and GET can be safely exposed to the adversary. On the one hand this is because the flag s is public information and on the other hand, because the adversary can't gain any advantage by setting the flag s to 1.

3. SSP proofs in practice

The creation of the SSP framework was motivated by the desire to better organize and understand large, composed protocol proofs. Since its creation, the SSP framework has been applied to a variety of large and small real-life protocols. Examples include Brzuska and Oechsner’s security proof of Yao’s garbling scheme [18] and Brzuska and Winkelmann’s security proof of a novel n -input PRF construction for proposed use in the Messaging Layer Security (MLS) key schedule [19].

This thesis contains four additional proofs that make use of the SSP framework: A security proof of the TLS 1.3 key schedule by Brzuska, Delignat-Lavaud, Egger, Fournet, Kohbrok and Kohlweiss (Publication II), a security proof of the key derivations (including the key schedule) of draft-11 of the MLS protocol by Brzuska, Cornelissen and Kohbrok (Publication III), as well as a security proof of a construction for rotating signature public keys by Cremers, Hale and Kohbrok (Publication IV). In this section, we discuss a few insights gained in those works.

3.1 Provability as a protocol design goal

With Publication II and Publication III this thesis contains two works that provide security proofs for key schedule designs. The former includes a proof of the security of the TLS 1.3 key schedule and the latter one of the MLS key schedule, as well as the MLS Ratcheting Tree. Even though MLS is the more complex protocol in terms of its provided feature set and the associated cryptographic guarantees, the security proof in Publication II required markedly more effort to produce, but also to present on paper in a concise manner, as at least partially evidenced by the difference in length of Publication II and Publication III. Thus, at least with respect to SSP, MLS has vastly better *provability* than TLS 1.3 for a variety of reasons that we discuss in Section 6 of Publication II. We will revisit two of those reasons here and contrast them with the protocol design of MLS. While we encountered these issues during our SSP-based proof, both issues likely

affect other proof approaches as well.

Lack of modularity. SSP excels at modular proofs, where each component can be isolated and idealized independently from the rest of the protocol, such that a series of local reasoning steps (as shown, for example, in Section 2.8.2) yields a security proof of the protocol as a whole. Thus, unsurprisingly, the biggest issue we faced in the TLS 1.3 security proof are instances where parts of the protocol were not sufficiently independent. For example, after computing the Diffie-Hellman secret (i.e. the g^{xy} group element resulting from the initiator's x and the receiver's y), the protocol immediately injects g^{xy} into the key schedule instead of first performing an extract operation. As a consequence, we were not able to reduce the resulting double-operation to a more self-contained assumption, forcing us to base our proof on a stronger, rather more complex and non-standard assumption (cf Section 3.4 of Publication II). A more modular approach, where the DH secret is first extracted followed by its injection into the schedule in a separate step, would have come at the cost of an additional derivation step, but would have resulted in an easier reduction into a set of better-understood assumptions.

Delayed domain separation. Another major issue was the lack of immediate domain separation of keys derived in some of the schedule's state transitions. The majority of issues here were caused by a missing separation between *application* and *resumption* PSKs, where application PSKs are meant to be injected at the start of a protocol run, whereas resumption PSKs can be stored by either party and later used to resume a previous protocol run. Both are injected in the same key schedule state and initially, the protocol doesn't distinguish between the two. The difference between the two types of PSK is made explicit only in the transcript, which is also injected into the key schedule as context, but much later in the key schedule.

This delay in distinguishing between the two scenarios (a freshly started protocol run and a resumption) prevents us from reasoning locally about the security achieved by injecting the PSK. Instead, it forces us to argue about the security properties of the whole sequence of state transitions (from injection of the PSK up until injection of the transcript), thus significantly complicating the proof.

If the PSK and the (partial) transcript were injected in the same transition, or even better, the PSKs were distinguished by an added derivation step before their injection that included their role (application or resumption) as a label, we could prove the properties of the output locally based on its inputs.

Importance of provability in protocol design. The result of these two quirks of the TLS 1.3 key schedule, as well as some other issues is a disproportionate increase in proof complexity, which results in turn in higher proof

effort, as well as greater size of proof presentation and increase in difficulty of proof readability. While a large portion of the protocol was proven secure using simple idealization steps as described in Section 2.8, the aforementioned quirks were almost exclusively responsible for the greater proof complexity.

In both cases described above, the cleanest way to solve the issue is adding an additional key derivation step, thus making the design more modular at the cost of an additional key derivation operation, which, in most cases, equates to three additional invocations of the hash function used in the particular run of the protocol.

The MLS working group has, for the most part, chosen to follow a more modular design, which results in better provability overall and, arguably, in a cleaner protocol design. An example co-authored by the author of this thesis is the pre-shared key (PSK) injection mechanism, where a unique label depending on the functionality provided by the PSK is upon injection into the rest of MLS' key schedule to ensure a clean separation of key material¹.

The formal analysis of MLS progressed quickly enough for released works to inform the protocol design process (despite the complexity of the protocol)². Publication III and Publication IV are two such works, along with [3], [31], and [12] to list just a few. On the one hand, this is the result of MLS being a green-field project with benefit of the experience from the TLS 1.3 design process. On the other hand, we were able to progress with the analysis of MLS much quicker due to the experience gained by analyzing the TLS 1.3 key schedule.

For a more thorough discussion of the concrete shortcomings of the two protocols (or their drafts) and the resulting difficulties during the proof effort, see Sections 1.1 and 6 of Publication II and Section III of Publication III respectively.

3.2 Handle construction and mapping

The quirks in the TLS 1.3 key schedule discussed in Section 3.1 lead to a less straight-forward proof in Publication II as compared to Publication III, even though at their heart, both proofs rely on the proof technique described in Section 2.8.

In this section, we briefly describe the *handle mapping* proof technique used in Publication II to deal with handle collisions.

¹See here for the pull request introducing the mechanism: <https://github.com/mlswg/mls-protocol/pull/336>

²Two examples of feedback that resulted from the work on Publication III are <https://github.com/mlswg/mls-protocol/pull/446> and <https://github.com/mlswg/mls-protocol/pull/453>.

Handling honest keys The TLS 1.3 keys schedule security proof relies on the same multi-instance approach based on handles as the PRF example in Section 2.7, where each key is stored by the game under a handle and the adversary (or any other package) calling an oracle provided by the core key schedule model (see Figure 11 of Publication II) can indicate the key they would like to interact with by using its handle. The TLS key schedule model is built such that the handle of a key encodes its creation (or derivation) history (see, e.g., Figure 9 in Publication II). For example, a key that resulted from the combination of two Diffie-Hellman shares X and Y might have the handle $\text{dh}\langle\text{sort}(X, Y)\rangle$. This handle construction scheme allows the game to keep track of keys and their history based on their handle.

However, this way of constructing handles potentially allows the adversary to have the game store the same key under two different handles. For example, computing a key from the pairs of Diffie-Hellman shares (X^a, Y) and (X, Y^a) will lead to the same key, but different handles. While not a problem on its own, it does lead to a problem as soon as the game idealizes key generation of the protocol's xtr operation (see Section 3.3 of Publication II).

The security assumption on xtr essentially states that it is a dual PRF, i.e., that its output key is indistinguishable from a randomly generated key as long as at least one of its input keys is *honest*, i.e. expected to be indistinguishable from random to the adversary. As in the example in Section 2, the game stores the resulting key in a **KEY** package under its handle. If the adversary can make the game store the same (dishonest) key under two handles in the xtr input **KEY** package, it can make the game combine each key (via xtr) with the same honest key. If the resulting keys are equal, the adversary knows that the game is not generating its output keys randomly, otherwise it knows that it is and can distinguish the games accordingly.

As a result of this quirk in the handle construction, the game needs to ensure that each key is stored exactly once before it can idealize the xtr game.

Mapping For this purpose, the proof includes a step, where a *mapping* package is introduced that intercepts all oracle queries from the adversary to the game without changing the functionality from the adversary's point of view. The mapping package allows the adversary to use colliding handles (i.e. pairs of handles pointing to the same key), but switches each colliding handle with its earlier counterpart. In Publication II see Figure 15 for the resulting composed game and Figure 14 for the definition of the mapping package. As a result of the mapping package introduction, there are no more handle collisions from the point of view of the game and the xtr game can be idealized following the proof pattern described in Section 2.8.

3.3 Models beyond the real-vs.-ideal paradigm

The running example in Section 2, as well as Publication II, Publication III and Publication V all model security in the real-vs.-ideal setting. However, some security notions such as existential unforgeability under chosen ciphertext attacks (EUF-CMA) or integrity of plaintext/ciphertext (INT-PTXT/CTXT) are *search-based*, meaning that instead of distinguishing two games, the adversary has to produce a value that matches certain criteria to win.

In Publication IV, we show that the SSP framework can also be used to prove security under a search-based security notion. In particular, we prove an instance of the rotating signature scheme (RSIG) protocol family (introduced in the same work) to be secure under a security notion derived from EUF-CMA.

Instances of the RSIG protocol family allow a signer to generate a signature key pair, use the private key to generate signatures and then rotate the key pair by creating an *update message*, which other parties (verifiers) can use to update their copy of the signer's public signature key. In the corresponding security model PCS-SIG (post-compromise secure signatures), the adversary can create a signer and a verifier, and have them sign and verify arbitrary messages, or create and process update messages respectively. In addition, the adversary can compromise the signer to obtain its current private signature key. The goal of the adversary is to forge a signature such that the verifier accepts it as valid for a given message in a situation, where the key currently used by the verifier is not considered *corrupt* and where the forged signature is not one that the adversary has obtained by having the signer create a signature over the associated message. In the spirit of the original definition of post-compromise security by Cohn-Gordon, Cremers and Garratt [21], a key is not considered corrupt either if the signer was never compromised, or if the verifier has processed at least one update message that was created by the signer after the most recent compromise.

This security notion in the style of EUF-CMA requires the adversary to find a correct value rather than distinguishing between a real and an ideal behaviour of a protocol. However, the security proof in Publication IV can still make use of the proof techniques outlined in Section 2.8, because the security notion represents a search problem, but is still modelled as a distinguishing game. We achieve this by performing a small transformation as, for example, described by Brzuska and Lipiäinen in the Crypto Companion (Section 1.1) [17]: To win the indistinguishability game, the adversary has to distinguish between two games, one with $b = 0$ and one with $b = 1$, where the behaviour of the two games is exactly the same, except that when the adversary submits the forgery, the game returns the bit b , thus allowing them to win the game with probability 1 as soon as

they have found a forgery.

The security proof for an RSIG protocol based on an EUF-CMA secure signature scheme can be outlined as the following series of game hops. First we extend the traditional EUF-CMA game with the capability for the adversary to corrupt the owner of the secret key. Second, we lift the resulting C(orrutable)EUF-CMA to a multi-instance setting (MI-CEUF-CMA) using a hybrid argument as outlined in Section 2.9. Finally, we create a modular reduction and proceed with the proof in the same fashion as shown in Section 2.8.

3.4 Summary and outlook

The SSP framework is meant as a tool that amends the code-based game-playing proofs by Bellare and Rogaway and that allows the prover to structure proofs and models in a modular way. The packages on which the SSP framework relies, along with the associated package-algebra facilitate convenient proof patterns and allow the prover to visualize models and proof steps using package-graphs.

Publication II and Publication III show that the SSP framework is useful to analyze the key schedules that have become common in more complex protocols and especially Publication II shows that it is flexible enough to deal with protocols that were not designed with security proofs in mind. Publication IV and Publication V provide evidence that the SSP framework is suitable for the treatment of other cryptographic constructions. Future work will show how useful it is in the context of more complex primitives such as authenticated key exchange, which would be the natural next step after the existing work on key schedules.

Since the SSP framework shares properties with simulation based frameworks such as Universal Composability [20] (UC, see Section 4), it would also be interesting to see if it can be applied to more multi-party computation or zero-knowledge proof constructions, for both of which, UC is commonly used in security proofs.

Works such as Publication V and the SSProve framework [2] represent proof that the SSP framework can be effectively applied to computer-aided cryptography tools (see Section 5 for a more in-depth discussion). Here, it will be interesting to see how well the SSP frameworks helps in scaling tools traditionally focused on smaller primitives such as EasyCrypt. Another option worth exploring would be a purpose-build tool that natively implements the package-algebra of the SSP framework, as well as its other differences from BR' code-based games. Such a tool could also visualize package compositions and proof steps using SSP's graph-based notation.

4. Compositional Frameworks

When Bellare and Rogaway introduced their code-based game-playing framework [10], they were not the only ones to recognize the need to tame the complexity of definitions and proofs. Several competing proof approaches and definitional frameworks were developed before and after the publication of BR’s framework for game-playing proofs, with a number of them taking a modular approach in a similar way as the SSP framework. A modular approach is meant to reduce the complexity of definitions and proofs of larger protocols and enables the re-use of individual components. In this section, we provide a brief overview over a small number of related frameworks with this focus and discuss how each framework relates to the basic SSP framework introduced in Section 2.

A conceptual parallel between SSP and all frameworks presented in this section is the underlying premise of the composition theorems that exist in each framework: That the individual components’ state is strictly separate, which implies that the computations performed by each component indeed depend only on the state of the (composed) component, as well as potential explicit inputs. This context-independence then gives rise to the ability to safely compose individual components.

For the comparison with the SSP framework, we focus on frameworks that have seen recent and extensive use in the analysis of larger protocols such as TLS and MLS. For a more complete overview over the history of compositional frameworks, see Appendix A of [20].

Framework categories. We divide frameworks into two categories as suggested by Maurer in Section 4.2 of [35]: those that follow a *top-down* approach and those that follow a *bottom-up* approach. We cover Universal Composability, Abstract Cryptography and Constructive Cryptography, all of which take the top-down approach and then turn to the bottom-up framework used in the book “The Joy of Cryptography”. At the end of this section, we compare the individual frameworks with the SSP framework.

4.1 Top-down frameworks

Top-down frameworks follow a clean-slate definitional approach, by defining higher level concepts such as parties, channels and distinguishers (i.e. classes of adversaries) for use in security definitions. At the same time, these frameworks are not necessarily designed with a particular proof technique in mind. All frameworks in this category follow a *simulation-based security* approach.

4.1.1 Simulation based security

Simulation based security is an alternative definitional approach to game-based security as described in Section 1. In contrast to game-based security, it defines security in terms of the *ideal functionality* of a protocol (as first used by Goldreich, Micali and Wigderson [37]).

The ideal functionality is a machine that maintains state and can compute and return values when called, both based on the inputs given when called and its internal state. The behaviour of the ideal functionality is at the core of the security definition for a given protocol. For example, the ideal functionality of an encryption protocol could take as input a message and return only the length of the resulting ciphertext.

In simulation based security, a computationally restricted adversary has to distinguish between two settings. In the first setting, the adversary interacts with the *parties* executing the protocol. The parties will typically keep secret state to compute their outputs to the adversary. In the second setting, the adversary instead communicates with a *simulator*. The simulator only has access to its own state and the protocol's ideal functionality, which it can use to compute its outputs to the adversary. A given protocol is secure w.r.t. an ideal functionality if there exists an efficient simulator such that an adversary can distinguish between the simulator and the real protocol execution only with negligible probability.

Continuing the example above, proving security for an encryption protocol would require proving the existence of a simulator, which can convincingly simulate the protocol execution given only the length of the ciphertext for a given message. When ciphertexts are indistinguishable from random strings of the same length, the simulator could provide random strings of suitable length as simulation of real ciphertexts.

SSP and simulation based security. Interestingly and perhaps surprisingly, the SSP framework can also be used for simulation-based security definitions (such as those using the UC framework) and proofs. An example of a simulation based security definition can be found in Publication II, where Section 4.5 defines key schedule security in a simulation-based way. The security proof follows the game-hopping style described in Section 2.8. The proof describes the simulator as a composition of packages that can

simulate the outputs of the protocol to the adversary interacting only with the protocol’s ideal functionality (see Fig. 26b of Publication II).

4.1.2 Universal Composability

Universal Composability (UC) as proposed by Canetti [20] is a widely used compositional framework that specifies a definition style for simulation-based security. The adversarial model in the UC framework is slightly different than the basic simulation-based security paradigm described in Section 4.1.1, as it introduces the *environment* as (additional) adversarial component. The environment provides the adversaries and the parties with potential initial values for the protocol execution and otherwise observes all outputs by the parties and the adversary. The environment can also interact with the adversary during the protocol execution.

In contrast to the basic simulation based security approach, the simulator has to simulate both the parties *and* the adversary to the environment. It is then the task of the environment to decide if it has just interacted with the simulator and the ideal functionality or the adversary and the parties.

In the context of composition, the environment models not just adversarial interaction, but also other potential protocol components that might provide inputs to the protocol in question. Thus, if a component is secure in the context of UC, it can be securely composed.

4.1.3 Abstract Cryptography

Introduced by Maurer and Renner [35], Abstract Cryptography formalizes components of secure protocols (channels, parties, distinguishers, etc.) in an abstract manner. The purpose of the abstractions is on the one hand to enable simple, easy to understand definitions, as well as to prove statements on high-level definitions that are then applicable to any lower-level constructions. Abstract Cryptography relies on the same real.-vs.-ideal paradigm as UC and also provides a composition theorem for its security notions.

The cryptographic algebra that facilitates the composition theorems for *systems* via their individual *interfaces* defines *resource systems* (resources) and *converter systems* (converters). A resource can be composed with a converter (given that their interfaces match), which results in a new resource system.

The security of a protocol can then, for example, be defined in terms of the advantage of an adversary (a distinguisher) in distinguishing two resources with the same interface.

4.1.4 Constructive Cryptography

Constructive Cryptography is an application of Abstract Cryptography by Maurer [34] to classic cryptographic primitives and protocols. It implements the concepts introduced in Abstract Cryptography such that traditional cryptographic proof methods can be applied with the goal of enabling a *constructive* approach, where, for example, a MAC is defined as a primitive that constructs an authenticated channel from a secret key and an unauthenticated channel. The notion of construction is composable, such that constructions can in turn be composed to yield new constructions.

4.1.5 SSP and top-down frameworks

With their focus on definitions rather than specific proof techniques, top-down frameworks are typically compatible with the proof approaches of frameworks such as SSP.

Computational model. The SSP framework and any of the top-down frameworks can be used in conjunction if one uses packages to define the systems/machines/programs (i.e. the computational model) that underlie the concepts of the respective framework. On the one hand, we thus benefit from the generality of the resulting definitions and the properties that lower-level constructs inherit from higher-level definitions in the context of a top-down framework. On the other hand, we can make use of SSP's package algebra in the concrete security proofs and, for example, use the proof patterns described in Section 2.8.

For example, consider the PRF example defined in Figure 2.5b in the context of the Abstract Cryptography framework. We can view a Key_{in} package as a resource system with two interfaces: One containing the GET oracle and one containing the SET oracle. The PRF package composed with the Key_{out} then takes the role of a converter system, which, composed with Key_{in} package again yields a resource system with the additional EVAL oracle as part of the output interface. We can now use the proof patterns described in Section 2.8 that make use of the SSP package algebra to conduct concrete security proofs regardless of the overarching definitional framework.

4.2 Bottom-up frameworks

In contrast to top-down frameworks, bottom-up frameworks, such as BR's code-based game-playing technique or the SSP framework, were designed as evolutions of existing approaches to definitions and proof techniques. We now briefly introduce the framework used in the Book "The Joy of Cryptography" [40] and compare it to the SSP framework.

4.2.1 “The Joy of Cryptography”

In his book “The Joy of Cryptography” [40], Rosulek introduces a framework (which we call *JC* for convenience) based on Bellare and Rogaway’s code-based game-playing framework and the associated game-hopping proof technique. The goal of the *JC* approach is primarily educational, as it is used, to the knowledge of the author, exclusively to conduct relatively simple example proofs in the context of the textbook.

In contrast to the frameworks detailed in Section 4.1, the *JC* framework represents more of a bottom-up approach, as it does not attempt to utilize layers of abstraction to generalize results or simplify definitions, but rather further formalizes existing definitional and proof techniques in general and the code-based, game-playing approach in particular. It is, however, a compositional framework, albeit at a lower level of abstraction, making it very useful in the context of proof writing even though it does not provide definitions of higher-level concepts such as parties or channels.

The *JC* framework follows a very similar approach as the *SSP* framework. It calls a game, which consists of a set of oracles, as well as the associated state a *library*, where each library has an interface that details the names, as well as input and output types of its oracles. Further, libraries have a number of properties. In particular, they can be composed, either with other libraries (yielding a *compound library*) or an adversary (yielding a *compound calling program*), with the composition operation being *associative*. The *JC* framework also defines relations between libraries, such as (computational) indistinguishability, as well as *interchangeability*, essentially indistinguishability with an adversarial advantage of 0.

Finally, the framework uses libraries to define security games and the associated library algebra to perform game-hopping proofs.

SSPs as an extension of the *JC* framework. A *JC* library, just as an *SSP* package, contains a set of oracles and their associated state variables. Similarly, a compound library corresponds to a set of sequentially composed packages.

The difference between the *JC* and the *SSP* framework is their scope. While the *SSP* framework is meant as a fully formalized framework for application to large and modular protocols, the *JC* framework is meant as an educational tool for a limited and well-defined number of example definitions and proofs. The consequence is that the *SSP* framework has a fully formalized package algebra supporting both sequential and parallel composition, while the *JC* framework only requires the sequential chaining lemma to perform (comparably) simple reduction proofs.

Summarizing, the *SSP* framework can be considered an extension of the *JC* framework, providing a more complete formalization, as well as the visual notation for composed packages.

5. The SSP framework and computer-aided cryptography

The SSP framework is (at the moment) primarily a methodology for pen-and-paper proofs. However, as discussed in Section 2.10.1, its origins lie in the field of computer-aided cryptography and since its inception it has been used in turn to guide computer-aided cryptographic proofs.

In this section, we discuss applications of the SSP framework to computer-aided cryptography. First, we discuss unfinished work that explores the use of relational F^* to implement the SSP framework (Section 5.1). We then summarize the SSProve [2] framework, a full formalization of the SSP framework using the Coq [42] proof assistant (Section 5.2). Finally, we discuss the findings of Publication V, where we use the SSP framework to guide proofs using the EasyCrypt [7] tool set (Section 5.3).

5.1 Relational F^*

Initial work has been undertaken to formalize SSP in relational F^* by Kohbrok, Kohlweiss, Ramananandro and Swamy (KKRS) [32], who use F^* to create setoids (types with associated equivalence relation) as introduced by Barthe, Capretta and Pons [6] to model SSP packages and their algebra. The work, although unfinished, describes an approach for package modelling in F^* that is different from the module based verification approach described in Section 2.10.1 in that it uses the F^* type system (instead of its module system) to model SSP packages. More concretely, KKRS use the F^* type system to define packages as setoids. Setoid-packages allow the explicit formalization of the SSP package algebra and reasoning about the relationships between individual packages (e.g. functional equivalence or ϵ -indistinguishability). The relational approach thus enables, for example, full reduction steps as described in Section 2.8, including the summation of adversarial advantages with each game hop.

While KKRS provide the scaffolding necessary for package composition, the work was halted due to difficulties in discharging high-level proof steps such as those shown in Section 2.8. Continuing this approach of

formalizing the SSP framework would be an interesting avenue for future work.

5.2 SSProve: SSP formalized in Coq

The first full formalization of the SSP framework for use with computer-aided cryptography tools was performed by Abate, Haselwarter, Rivas, Van Muylder, Winterhalter, Hritcu, Maillard and Spitters [2] (SSProve authors). Their SSProve framework models the full package algebra, including probabilistic reasoning about individual game hops in a computer-verifiable way and extends the original pen-and-paper formalization of Publication I with formal semantics of the language used to define oracles in the Coq framework [42].

SSProve also allows the combination of the high-level proof structure enabled by the SSP framework with machine-checked equivalence proofs, which, as described in Step 3 in Section 2.8.2 represent a major difficulty when dealing with large models on paper.

Along with the formalization, the SSProve work [2] presents several examples showing that SSProve can be used to combine SSP-style proofs with lower-level probabilistic reasoning about cryptographic primitives.

In particular, they implement (and formally prove) the KEM-DEM example presented in the introduction and Section 4 of Publication I. While writing the formal proof, the SSProve authors discovered a flaw in the original pen-and-paper proof.

The machine-check finding a previously overlooked flaw in our pen-and-paper proof clearly demonstrates the importance of computer aided cryptography in general and of works like SSProve in particular. Since all computer-aided proofs that make use of the SSP framework have been for smaller protocols, it will be interesting to see if computer-aided cryptography can indeed benefit from the SSP framework's compositional properties when applied to larger protocols.

More recently, Haselwarter, Hvass, Hansen, Winterhalter, Hritcu and Spitters have gone a step further and connected SSProve to HACSpec [36], an emergent protocol specification language, thus, in the same vein as the miTLS project [24], moving formal security proofs closer to executable protocol implementations.

5.3 Using SSP to guide EasyCrypt proofs

In Publication V, we explore the use of the SSP framework to guide security proofs of composed protocols in EasyCrypt. In contrast to SSProve, in this work we do not fully formalize the SSP framework but instead introduce a

mapping from individual SSP concepts to EasyCrypt. Using this mapping, we provide a security proof for NaCl’s Cryptobox [11].

Publication V yields several insights into the design of SSP, its application to EasyCrypt, as well as some guidance for the design of a potential SSP-specific tool. We briefly summarize the findings here. For a complete discussion, see Section 8 of Publication V.

5.3.1 Packages in EasyCrypt

In Publication V we use EasyCrypt module types to define package interfaces. While package interfaces only contain oracle names, EasyCrypt module types allow us to be more precise by letting us define EasyCrypt procedure types, which additionally describe the nature of the oracles’ input and outputs.

A package that implements a given interface is then defined as an EasyCrypt module of the corresponding module type. EasyCrypt modules in turn define the oracles (EasyCrypt procedures) and state variables of the package.

Finally, the input interface of a package is expressed by adding the type of another module as a module’s parameters. We can then create a composed module by parameterizing the module and another module with a matching interface.

When using EasyCrypt modules to represent SSP packages, there is no distinction between packages and package instances outside of the module name. However, EasyCrypt does allow the re-use of modules definitions in the definition of other modules. As a result, we can define a module as a package definition and then create an arbitrary number of package instances by essentially cloning that module. For more concrete examples of SSP packages in EasyCrypt, see Publication V.

Overall and despite some problems with state management which we discuss in Section 5.3.3, the EasyCrypt module system is a good fit for SSP’s packages.

```

module type KEYout =
  proc set (_ : handle, _ : key): unit
  proc get (_ : handle): key

module KEY0 =
  var K : handle → key

  proc set (h : handle, k : key) =
    if (KEY0.K.[h] ≠ ⊥)
      return ;
    KEY0.K.[h] ← k;

  proc get (k : handle) =
    return KEY0.K.[h]

```

Figure 5.1. Example definition for the Key^0 package, as well as its output interface using the EasyCrypt module system. Note that the *get* procedure simply returns \perp when there is no key for the given handle h . See Section 5.3.2 for more details on how we implement **assert** in Publication V.

5.3.2 Flexibility in assert semantics

As discussed in Sections 2.6 a and 2.10.3, there are multiple ways of implementing the semantics of asserts. SSProve, for example, specifies **assert** to restrict the class of adversaries to the one not triggering any assertions. As a more general purpose proof assistant, EasyCrypt doesn't have a specific assert semantic. On the one hand, this means that the prover has to implement their own assert functionality as we did for the proof of Cryptobox. On the other hand, the prover is free to choose an assert semantic that fits their modelling needs (cf. Section 2.6).

In the Cryptobox security proof, we implemented an assert semantic, where triggering an assert means that the currently running oracle returns a special symbol. Other oracles in the call chain receiving this symbol can then revert any state-changes made during this particular oracle call and pass on the symbol until it reaches the adversary. One could also implement an assert semantic that mimics oracle silencing [39] as described in Section 2.6.

The flexibility afforded by EasyCrypt is beneficial in that it provides a larger degree of flexibility and forces the prover to consider the available options. However, it also forces them to manually implement the chosen assert semantic. A good middle path for a new, SSP-specific tool might be to provide the most commonly used assert styles out-of-the box.

5.3.3 Package state

While EasyCrypt modules are generally a good fit as a model for SSP packages, its memory model is harder to adapt to the needs of the SSP framework. In particular state-separation and state initialization are not straight forward.

State-separation. In all lemmas in Publication V, state-separation between individual EasyCrypt modules needs to be mandated explicitly. This is because EasyCrypt follows a definitional approach, where an overarching experiment controls all modules (including their state) and instantiates the adversary. For this proof approach, the framework needs to access the state of the modules directly, violating the state-separation principle required by the package algebra of the SSP framework. It follows that for each lemma formulated in EasyCrypt, we need to explicitly reason about the state of all modules involved, e.g. stating that a given adversary does not access the state of composed modules directly. In practice, however, this is less of a fundamental problem, as EasyCrypt provides a convenient way to express such restrictions in lemmas.

State initialization. The difficulties with state initialization also stem from EasyCrypt's design assuming the existence of an experiment that

initializes the state of all modules before instantiating the adversary. In contrast, in the SSP framework all state variables are initialized to a well-specified default value. For more elaborate initialization procedures (i.e. assigning a specific non-default value to a given variable), SSP can (via asserts, see Section 2.6) force the adversary to call an initialization oracle before accessing other oracles that rely on any initialized values.

Since state initialization *has* to happen explicitly in EasyCrypt, all lemmas have to be explicit about the initial state of all packages over which they quantify. In Publication V we achieve this explicit initialization of state by locally inserting initialization code where needed.

Both difficulties regarding state management spring from the fact that EasyCrypt was ultimately designed for a different approach to modelling cryptographic security. This style of proof can be emulated using SSP by having a package implementing the algorithm of the experiment. The experiment package then calls the adversary which in turn is composed with the rest of the model. However, since the proof patterns described in Section 2.8 rely on the fact that the adversary is the left-most package in the composition, most SSP-based proofs will likely be written in a style without such an experiment package.

Bibliography

- [1] Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of cryptography*, 15(2):103–127, 2002.
- [2] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Nikolaj Sidorenko, Catalin Hritcu, Kenji Mailard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. Cryptology ePrint Archive, Report 2021/397, 2021. <https://ia.cr/2021/397>.
- [3] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21*, page 1463–1483, New York, NY, USA, 2021. Association for Computing Machinery.
- [4] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795, 2021.
- [5] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-20, Internet Engineering Task Force, March 2023. <https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-20>.
- [6] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.
- [7] Gilles Barthe, Benjamin Grégoire, Sylvain Héraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 71–90, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [8] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. Subtleties in the definition of ind-cca: when and how should challenge decryption be disallowed? *Journal of Cryptology*, 28(1):29–48, 2015.
- [9] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.
- [10] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 409–426, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

- [11] Daniel J Bernstein. Cryptography in NaCl. *Networking and Cryptography library*, 3:385, 2009.
- [12] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018. <https://hal.inria.fr/hal-02425247>.
- [13] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the tls 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, 2017.
- [14] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the tls handshake secure (as it is). In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 235–255, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [15] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. mitls: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy*, 14(6):18–25, 2016.
- [16] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.
- [17] Chris Brzuska and Valteri Lipiäinen. Companion to cryptographic primitives, protocols and proofs, November 2022. <https://cryptocompanion.github.io/cryptocompanion/cryptocompanion.pdf>.
- [18] Chris Brzuska and Sabine Oechsner. A state-separating proof for yao’s garbling scheme. Cryptology ePrint Archive, Report 2021/1453, 2021. <https://ia.cr/2021/1453>.
- [19] Chris Brzuska and Jan Winkelmann. Nprfs and their application to message layer security, 2020. <http://chrisbrzuska.de/2020-NPRF.html>.
- [20] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. <https://ia.cr/2000/067>.
- [21] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, 2016.
- [22] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS ’17*, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery.
- [23] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of tls 1.3: 0-rtt, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485, 2016.
- [24] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. miTLS: A Verified Reference Implementation of TLS. <https://www.mitls.org/>.

- [25] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the tls 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 463–482, 2017.
- [26] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 341–350, 2011.
- [27] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, USA, 2000.
- [28] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.
- [29] Google. Google Transparency Report. Technical report, January 2022. <https://transparencyreport.google.com/https/overview?hl=en>.
- [30] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. <https://ia.cr/2005/181>.
- [31] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 268–284, 2021.
- [32] Markulf Kohlweiss, Konrad Kohbrok, Tahina Ramananandro, and Nikhil Swamy. Relational F^* for state separating cryptographic proofs. https://github.com/FStarLang/FStar/wiki/Relational-F*-for-State-Separating-Cryptographic-Proofs (accessed: 03.03.2022), 2020.
- [33] Moxie Marlinspike and Trevor Perrin. The Signal Protocol. Technical report, November 2016. <https://signal.org/docs/>.
- [34] Ueli Maurer. Constructive cryptography – a new paradigm for security definitions and proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications*, pages 33–56, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [35] Ueli Maurer and Renato Renner. Abstract cryptography. In Bernard Chazelle, editor, *The Second Symposium on Innovations in Computer Science, ICS 2011*, pages 1–21. Tsinghua University Press, 1 2011.
- [36] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Technical report, Inria, March 2021. <https://hal.inria.fr/hal-03176482>.
- [37] Silvio Micali, Oded Goldreich, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, pages 218–229. ACM, 1987.
- [38] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. <https://www.rfc-editor.org/info/rfc8446>.
- [39] Phillip Rogaway and Yusi Zhang. Simplifying game-based definitions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 3–32, Cham, 2018. Springer International Publishing.
- [40] Mike Rosulek. *The Joy of Cryptography*. <https://joyofcryptography.com>.

Bibliography

- [41] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *Cryptology ePrint Archive, Report 2004/332*, 2004. <https://ia.cr/2004/332>.
- [42] The Coq Development Team. The coq proof assistant. *Technical report, October 2019*. <https://doi.org/10.5281/zenodo.3476303>.



ISBN 978-952-64-1355-6 (printed)
ISBN 978-952-64-1356-3 (pdf)
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science
www.aalto.fi

**BUSINESS +
ECONOMY**

**ART +
DESIGN +
ARCHITECTURE**

**SCIENCE +
TECHNOLOGY**

CROSSOVER

**DOCTORAL
THESES**