PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'ENS Paris et l'Inria Paris

# Preuves mécanisées de protocoles cryptographiques et leur lien avec des implémentations vérifiées


Soutenue par
**Benjamin LIPP**
Le 28 juin 2022


École doctorale n°386
**Sciences Mathématiques de Paris Centre**

Spécialité
**Informatique**

Composition du jury :

| | |
|---|---|
| David POINTCHEVAL<br>Directeur de recherche, CNRS | *Président* |
| Véronique CORTIER<br>Directrice de recherche, CNRS | *Rapporteure* |
| Ralf KÜSTERS<br>Professeur, Université de Stuttgart | *Rapporteur* |
| Cédric FOURNET<br>Senior Principal Research Manager, Microsoft Research | *Examinateur* |
| Bruno BLANCHET<br>Directeur de recherche, Inria | *Directeur de thèse* |
| Karthikeyan BHARGAVAN<br>Directeur de recherche, Inria | *Directeur de thèse* |

ENS | PSL

Inria

# Mechanized Cryptographic Proofs of Protocols and their Link with Verified Implementations

## Colophon

This thesis was typeset using LaTeX and the `memoir` documentclass. It is based on Friedrich Wiemer' thesis *Security Arguments and Tool-Based Design of Block Ciphers*[1], itself based on Aaron Turon's thesis *Understanding and expressing scalable concurrency* [2], itself a mixture of `classicthesis`[3] by André Miede and `tufte-latex`[4], based on Edward Tufte's *Beautiful Evidence*.

The bibliography was processed by Biblatex.

The body text is set 10/14pt (long primer) on a 26pc measure. The margin text is set 8/9pt (brevier) on a 12pc measure. Matthew Carter's Charter acts as both the text and display typeface. Monospaced text uses Jim Lyles's `Bitstream Vera Mono` ("Bera Mono").

[1] https://asante.dev/publication/phd_thesis/, source code available at https://github.com/pfasante/phd_thesis.

[2] https://people.mpi-sws.org/~turon/turon-thesis.pdf

[3] https://bitbucket.org/amiede/classicthesis/

[4] https://github.com/Tufte-LaTeX/tufte-latex

# *Acknowledgements*

My time in the Prosecco research group at Inria Paris has been enormously rich for me – in personal and academic growth, in the colleagues I met, and the friends I made. I am very grateful to my PhD advisors Bruno and Karthik, and thank them wholeheartedly for giving me this opportunity, funding my PhD, and for guiding me through this adventure, by offering challenging projects, with patience and vision. Thank you for being there in countless meetings, for discussing my work, my ideas, and for navigating any difficult situations. I also thank you for offering me the freedom to travel to a summer school, a student program committee meeting, and many workshops and conferences.

I thank Véronique Cortier and Ralf Küsters for having served as reviewers of my thesis, and for being part of the jury of my defence. I thank Cédric Fournet and David Pointcheval for being part of the jury, and David Pointcheval for being president of the jury.

I am grateful to Bruno for carefully proofreading my thesis manuscript through multiple iterations and helping me find ways to present my work, to Karthik for providing valuable feedback on the manuscript, and to Anna for her helpful comments on the cv2fstar chapter. I thank Aymeric, Bruno, Christian, Denis, Goutam, Itsaka, and Rob for their comments and advice on my rehearsal defence talk.

Scientifically, this journey started with Bruno teaching me how to use the CryptoVerif proof assistant during my master internship. This has certainly been the right way for me to get into the field of provable security. Before this initiation to CryptoVerif, I did not see myself writing cryptographic proofs by hand; now, I feel ready for that, too. Although, for manual proofs, I am missing CryptoVerif's last output line in case of success, `All queries proved.` Besides its comforting guarantees, it has some addiction potential. Sometimes, when I used the tool in some strange ways that made it crash, CryptoVerif would tell me to contact a certain `Bruno.Blanchet@inria.fr`, and to `[p]lease report [a] bug [...] including input file and output`. For this, I was of course in the greatly privileged position of being able to just walk over three offices. Thus, Bruno, many thanks for patiently initiating me to mechanized cryptographic proofs, and for implementing some of my usability improvement ideas into the tool over the last years.

The other proof-oriented development environment in which I have been spending a lot of time is the F⋆ ecosystem. I want to thank everyone who helped me find my ways, and who welcomed me in the community, especially: Karthik, for some insightful pair programming sessions; Ben, who helped me navigate the voodoo sequence of commands needed to get Project Everest running, and who spent multiple pair programming sessions with me; Anna for helping to debug P-256 in my HACL⋆ setup; Marina for help with using HACL⋆; my office mate Denis for answering everyday F⋆ questions; Son and Vincent for providing helpful code snippets; Aymeric for explaining to me the version of the HPKE specification that I built upon;

Victor for help with HACL*'s OCaml interface; and Catalin for an insightful discussion about my cv2fstar work. I also thank those who answered my questions on the various online channels, in particular Cezar, Jonathan, Nik, and Tahina; and finally, Cédric, for organizing the Project Everest all-hands meeting that I attended in Cambridge.

I spent a long part of my PhD on the Hybrid Public Key Encryption standardization effort, and I am very happy and grateful that I could be part of it. I thank Karthik for bringing me into this project, and the other RFC co-authors Chris Wood and Richard Barnes for their interest in my provable security work, and their openness to my suggestions that came out of this work. I thank Chris for working on a preprint with me, and asking me if I want to be an official co-author of the RFC. It was a pleasure to work with you and the CFRG community, and I thank everyone who was involved in bringing this RFC to publication as RFC 9180. I am glad that the HPKE project also led me to my external co-authors Eike, Joël, Doreen, and Eduard, with whom, together with Bruno, we published a paper containing mechanized cryptographic proofs at Eurocrypt. I learned a lot from you about cryptographic proofs and how to present them, and I am very grateful for our collaboration. Thank you! Finally, I thank Franziskus for helpfully pushing me through finishing writing my blog post about HPKE, which turned out to be pretty effective on science Twitter.

During two years of my PhD, I was a teaching assistant at the university that is now called Université Paris Cité, and I am thankful for the teaching experience that I could gather in this time. I thank everyone who made this possible, especially Thomas Béraud, Ralf Treinen, Michele Pagani, and Cristina Sirangelo; the colleagues who helped me navigate the bureaucratic jungle coming with this, especially Glen; Pablo who administered our teaching Discord server during lockdown time, and Sam who found an online programming environment suitable for teaching.

I want to thank Véronique, Catalin, Bruno, Gilles, Karthik, and the PETS community for allowing me to gain paper reviewing experience as an external reviewer for individual papers. I also had the opportunity to participate in the shadow program committee of IEEE's Security&Privacy 2020 conference, which was an insightful experience that I am grateful for, and I thank Tom Moyer for organizing it.

I thank Rakesh Bobba, David Evans, and Cas Cremers for insightful conversations during mentoring sessions at online conferences.

Although it dates back to my master internship in Prosecco starting end of the year 2017, I want to repeat my gratefulness to Julian Herr, Mario Strefler, Mathieu Goessens, Jeff Burdges, and Harry Halpin, who helped me make my way to the Prosecco research group.

At Inria Paris and in Prosecco, I had an amazing time with great colleagues, and I want to thank you: Denis, my long-time office mate, for helping me with French bureaucracy and explaining French politics with a passion, and also for great dinners and French theater; Carmine, my other long-time office mate, for almost always beating me to arriving early in the office; Ben, for great discussions about cryptographic protocols; Rob, for valuable feedback on practice talks; Catalin, for organizing many Prosecco pub evenings, inviting to brunches, and helpful advice about academia;

Antonin, for asking me why I do cryptographic research and thereby making me remember Rogaway's essay that I read for the first time several years ago; Aymeric, for pointing me to the Science of Security survey paper; Prasad, for great advice, hikes, dinners, and working sessions on contact tracing; Victor, for sadly only one after-work drink as neighbors in the 13th before I moved to Bochum; Anna and Marina, for organizing a board game evening and many pub evenings; Kenji, for introducing cake Tuesday; Nadim, for his magical appearance on multiple occasions when I needed advice on new computer hardware; and Aaron, Adrien, Alain, Bruno, Florian, Iness, Jérémy, Justine, Karthik, Kristina, Marc, Paul-Nicolas, Ram, Son, Théo, Théophile, and Vincent, for the daily encouragement and distractions that made the time worthwhile and comfortable. I thank our team assistant Mathieu Mourey, for his great support, executed with ease and pragmatism, in all things administration and bureaucracy, especially for the travel expense reports of my long-distance train travels. Outside of Prosecco, I thank Léo for giving me insights into the politics of science, Inria, and IACR, and for helping me maintain my bike. I thank Sébastien, Matthieu, Juliette, Ivan, Yohan, Mauricio, and Nastassia who started the local sustainability group at Inria Paris together with me.

I want to thank Sylvain, a friend I made through meetings at multiple conferences, for his socio-anthropologic interest in security research and the stimulating conversations about it. I want to thank Karolin for interesting discussions about WireGuard and academia, offering refreshing point-of-views from a different angle.

For the last science-ish thank you, of course, not to forget the co-authors of the Eurocrypt 2021 rump session talk about the $(\alpha, \beta)$-technique that won us the rump session award: Phillip, Benedikt, Christoph, Doreen, Eduard, Fabian, Jonas, and Noemi.

I thank the developer aegis and the kind community of the Talon voice control and hands-free input software. You helped me bridge the time of recovery from my injury by providing a technologically astonishing software ecosystem. Without Talon, this thesis would certainly have been delayed some more or have less content.

Besides the academic parts of my life in Paris, I am happy to have enjoyed the company of many friends throughout the time of my PhD. I enjoyed a lot being part of the choir of Cité Internationale Universitaire de Paris; singing La Marseillaise with a large choir and orchestra in La Philharmonie de Paris is probably a level of integration that not many non-French people have the privilege to experience. Thank you Beate, Franzi, Josephine, Romy, Sahita, Marine, Gregory, Pierre, Julian, Antoine, and Olivier, to name only a few of my fellow choristes. I want to thank my friends who visited me in Paris physically or virtually: Felix and Jenny, Jakob, Julian and Evi, Vicky and Johannes, Xandra, Julian, Nico, Max, Nora, Nora and Johannes and Fiona and Clemens, Bogdan, Astrid, David, Nefta and Christian, Josi and Ekrem, Jens, Julian, ChrKr, and Bölz, Gunnar, Joachim, Anke, Sonja, Julia, and Harry. I want to thank Julian for our mostly perfectly asynchronous Signal chats; and Miriam, Marcel, and Lukas, for being there in our infamous Signal group. I am deeply grateful to my former flatmate Laura, and happy that we found a nice apartment in the 13th arrondissement together. Thank

you for teaching me many useful psychological and philosophical tools. I thank my then-following but now also former flatmate Franziska for bearing with me when I was mostly focussing on thesis work during the last months in Paris.

Finally, I want to express my gratitude to my parents Pia and Bernhard, who have been supporting me without hesitation on this journey, also financially. Without this I would not have been able to afford living in the 13th arrondissement of Paris for a good part of my PhD. I thank my sister Beatrice for always having an open ear and checking in on me. I am grateful that my grandparents Ernst and Johanna have supported me in all the journeys I took during my bachelor, master, and PhD studies. And finally, I am grateful that I could go this way together with my girlfriend Yvette, who has been a constant source of support and encouragement.

# *Abstract*

Cryptographic protocols are one of the foundations for the trust people put in computer systems nowadays, be it online banking, any web or cloud services, or secure messaging. One of the best theoretical assurances for cryptographic protocol security is reached through proofs in the computational model. Writing such proofs is prone to subtle errors that can lead to invalidation of the security guarantees and, thus, to undesired security breaches. Proof assistants strive to improve this situation, have got traction, and have increasingly been used to analyse important real-world protocols and to inform their development. Writing proofs using such assistants requires a substantial amount of work. It is an ongoing endeavour to extend their scope through, for example, more automation and detailed modelling of cryptographic building blocks. This thesis shows on the example of the CryptoVerif proof assistant and two case studies, that mechanized cryptographic proofs are practicable and useful in analysing and designing complex real-world protocols.

The first case study is on the free and open source Virtual Private Network (VPN) protocol WireGuard that has recently found its way into the Linux kernel. We contribute proofs for several properties that are typical for secure channel protocols. Furthermore, we extend CryptoVerif with a model of unprecedented detail of the popular Diffie-Hellman group Curve25519 used in WireGuard.

The second case study is on the new Internet standard Hybrid Public Key Encryption (HPKE), that has already been picked up for use in a privacy-enhancing extension of the TLS protocol (ECH), and in the Messaging Layer Security secure group messaging protocol. We accompanied the development of this standard from its early stages with comprehensive formal cryptographic analysis. We provided constructive feedback that led to significant improvements in its cryptographic design. Eventually, we became an official co-author. We conduct a detailed cryptographic analysis of one of HPKE's modes, published at Eurocrypt 2021, an encouraging step forward to make mechanized cryptographic proofs more accessible to the broader cryptographic community.

The third contribution of this thesis is of methodological nature. For practical purposes, security of implementations of cryptographic protocols is crucial. However, there is frequently a gap between a cryptographic security analysis and an implementation that have both been based on a protocol specification: no formal guarantee exists that the two interpretations of the specification match, and thus, it is unclear if the executable implementation has the guarantees proved by the cryptographic analysis. In this thesis, we close this gap for proofs written in CryptoVerif and implementations written in F$^\star$. We develop cv2fstar, a compiler from CryptoVerif models to executable F$^\star$ specifications using the HACL$^\star$ verified cryptographic library as backend. cv2fstar translates non-cryptographic assumptions about, e.g., message formats, from the CryptoVerif model to F$^\star$ lemmas. This allows to

prove these assumptions for the specific implementation, further deepening the formal link between the two analysis frameworks. We showcase cv2fstar on the example of the Needham-Schroeder-Lowe protocol. cv2fstar connects CryptoVerif to the large F$^\star$ ecosystem, eventually allowing to formally guarantee cryptographic properties on verified, efficient low-level code.

# *Résumé*

Les protocoles cryptographiques sont l'un des fondements de la confiance que la société accorde aujourd'hui aux systèmes informatiques, qu'il s'agisse de la banque en ligne, d'un service web, ou de la messagerie sécurisée. Une façon d'obtenir des garanties théoriques fortes sur la sécurité des protocoles cryptographiques est de les prouver dans le modèle calculatoire. L'écriture de ces preuves est délicate : des erreurs subtiles peuvent entraîner l'invalidation des garanties de sécurité et, par conséquent, des failles de sécurité. Les assistants de preuve visent à améliorer cette situation. Ils ont gagné en popularité et ont été de plus en plus utilisés pour analyser des protocoles importants du monde réel, et pour contribuer à leur développement. L'écriture de preuves à l'aide de tels assistants nécessite une quantité substantielle de travail. Un effort continu est nécessaire pour étendre leur champ d'application, par exemple, par une automatisation plus poussée et une modélisation plus détaillée des primitives cryptographiques. Cette thèse montre sur l'exemple de l'assistant de preuve CryptoVerif et deux études de cas, que les preuves cryptographiques mécanisées sont praticables et utiles pour analyser et concevoir des protocoles complexes du monde réel.

La première étude de cas porte sur le protocole de réseau virtuel privé (VPN) libre et open source WireGuard qui a récemment été intégré au noyau Linux. Nous contribuons des preuves pour plusieurs propriétés typiques des protocoles de canaux sécurisés. En outre, nous étendons CryptoVerif avec un modèle d'un niveau de détail sans précédent du groupe Diffie-Hellman populaire Curve25519 utilisé dans WireGuard.

La deuxième étude de cas porte sur la nouvelle norme Internet Hybrid Public Key Encryption (HPKE), qui est déjà utilisée dans une extension du protocole TLS destinée à améliorer la protection de la vie privée (ECH), et dans Messaging Layer Security, un protocole de messagerie de groupe sécurisée. Nous avons accompagné le développement de cette norme dès les premiers stades avec une analyse cryptographique formelle. Nous avons fourni des commentaires constructifs ce qui a conduit à des améliorations significatives dans sa conception cryptographique. Finalement, nous sommes devenus un co-auteur officiel. Nous effectuons une analyse cryptographique détaillée de l'un des modes de HPKE, publiée à Eurocrypt 2021, un pas encourageant pour rendre les preuves cryptographiques mécanisées plus accessibles à la communauté des cryptographes.

La troisième contribution de cette thèse est de nature méthodologique. Pour des utilisations pratiques, la sécurité des implémentations de protocoles cryptographiques est cruciale. Cependant, il y a souvent un écart entre l'analyse de la sécurité cryptographique et l'implémentation, tous les deux basées sur la même spécification d'un protocole : il n'existe pas de garantie formelle que les deux interprétations de la spécification correspondent, et donc, il n'est pas clair si l'implémentation exécutable a les garanties prouvées par l'analyse cryptographique. Dans cette thèse, nous comblons cet écart pour les preuves écrites en CryptoVerif et les implémentations écrites en F*.

Nous développons cv2fstar, un compilateur de modèles CryptoVerif vers des spécifications exécutables F⋆ en utilisant la bibliothèque cryptographique vérifiée HACL⋆ comme fournisseur de primitives cryptographiques. cv2fstar traduit les hypothèses non cryptographiques concernant, par exemple, les formats de messages, du modèle CryptoVerif vers des lemmes F⋆. Cela permet de prouver ces hypothèses pour l'implémentation spécifique, ce qui approfondit le lien formel entre les deux cadres d'analyse. Nous présentons cv2fstar sur l'exemple du protocole Needham-Schroeder-Lowe. cv2fstar connecte CryptoVerif au grand écosystème F⋆, permettant finalement de garantir formellement des propriétés cryptographiques sur du code de bas niveau efficace vérifié.

# Contents

# Part I

# PROLOGUE

# 1

# *Introduction*

Cryptology is a fascinating science, concerned with security and privacy of information, communication, and computation in the presence of adversaries. [1] Cryptographic systems form an important pillar of security and privacy of the digital aspects of our life styles, organisations, and governments: They are a technological basis for the trust we put into, e. g., web or cloud services, online banking, software updates for connected devices, and secure messaging. This last example, secure messaging, is especially important for groups susceptible to surveillance, like journalists, activists, and whistleblowers.

[1] Adapted from the website of the International Association for Cryptologic Research (IACR), https://www.iacr.org/.

Work in the domain of cryptology and the resulting cryptographic systems are inherently political, because of their ability to implement power structures, as laid out in Rogaway's essay "The Moral Character of Cryptographic Work" published in the year 2015 [Rog15]. He states:

[Rog15] Rogaway, *The Moral Character of Cryptographic Work*

> I am [...] concerned with what surveillance does to society and human rights. Totalized surveillance vastly diminishes the possibility of effective political dissent. And without dissent, social progress is unlikely.

Recent examples where a cryptographic protocol being developed and deployed (or planned to be deployed) provoked intense discussions and worries of being yet another hard-to-revert first step to ubiquitous surveillance are pandemic contact tracing [RBS20] and Apple's plans to introduce client-side scanning on smartphones to detect child sexually abuse material (CSAM). [2] In his essay, Rogaway calls for the cryptographic community to engage in anti-surveillance research and suggests several avenues to do so. One of them is *practice-oriented provable security*, and it is this area to which this thesis mainly aims to contribute. The term *provable security* stands for work on the strongest kind of assurance of cryptographic security that one can hope to achieve for the specification level of a cryptosystem. In practice, this kind of security is obtained by reduction proofs in the computational model. However, the relevance of theorems established by such proofs for actual real-world systems depends on the assumptions and abstractions they use. A common critique summarized by a survey on the *Science of Security* [Hv17] is that assumptions and abstractions are too far from reality such that the analyses' applicability to real-world cryptographic systems is rather limited. Practice-oriented provable security strives to use assumptions and abstractions that are as close to reality as possible, and aims to enable

[RBS20] Reichert et al., *A Survey of Automatic Contact Tracing Approaches Using Bluetooth Low Energy*

[2] https://www.theverge.com/2021/8/6/22613365/apple-icloud-csam-scanning-whatsapp-surveillance-reactions

[Hv17] Herley and van Oorschot, "SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit"

useful assessments of, e. g. key sizes and number of users for practical cryptographic systems. This requires modelling cryptosystems on a low-level and with a higher detail, which is tedious and error prone to do by hand; we present our suggestion to approach this issue in a bit.

Providing a proof along with the presentation of a new cryptographic protocol was not always standard. The Needham-Schroeder authentication protocol published in 1978 [NS78] did not have a security proof. In 1989, a proof was published [BAN90], but years later, in 1995, an attack was found using formal methods [Low95; Low96]. Nowadays, *game-based* proofs are broadly viewed as a means to facilitate cryptographic security proofs [Sho04; BR06]. However, with complex protocols like TLS [Res18] and the WireGuard VPN protocol published in 2017 [Don17], and the goal to have highly-detailed models, game-based proofs tend to get long and complex, which makes it hard to proofread them. In 2004, Bellare and Rogaway use clear words to describe how they perceive the situation [BR06]:

> In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor.

Halevi, in his essay "A plausible approach to computer-aided cryptographic proofs" [Hal05], adds:

> Some of the reasons for this problem are social (e. g., we mostly publish in conferences rather than journals), but the true cause of it is that our proofs are truly complex.

Finally, Bellare and Rogaway agree with Halevi by repeating his call for the creation of "automated tools, that can help write and verify game-based proofs" [Hal05; BR06].

In this thesis, we propose to tackle the issue of practice-oriented provable security and highly detailed models by increasing the applicability and visibility of computer-aided proofs. We focus on real-world protocols. Our hypothesis is that cryptographic proofs of practical usefulness, are feasible using automated proof assistants, provide a high assurance that the theorems are correct, and are a useful part of protocol development.

Before introducing the computational model of cryptography, we want to say that computer-aided analysis of cryptographic protocols is mainly conducted in two conceptually different models, the *symbolic* model and the *computational* model. Both models consider an active adversary, which controls the network and can observe, duplicate, suppress, as well as construct new messages from messages it observed, and inject them. In the symbolic model, cryptographic building blocks [3] are considered perfect and modeled as function symbols, and messages are terms on these symbols. This means the adversary cannot exploit any internal structure of terms like public keys or ciphertexts. For example, a particular ciphertext can only be decrypted if exactly the key used for encryption is used in the decryption function. The adversary in the symbolic model is a Dolev-Yao adversary [DY83]. In terms of computation, it is restricted to the functions that are available within the protocol model. This defines explicitly what the adversary *can* do. In the symbolic model, logical protocol flaws and protocol traces leading to them, and thus to the violation of a security property can be found. The

[NS78] Needham and Schroeder, "Using encryption for authentication in large networks of computers"

[BAN90] Burrows et al., "A Logic of Authentication"

[Low95] Lowe, "An attack on the Needham-Schroeder public-key authentication protocol"

[Low96] Lowe, "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR"

[Sho04] Shoup, *Sequences of games: a tool for taming complexity in security proofs*

[BR06] Bellare and Rogaway, "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs"

[Res18] Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*

[Don17] Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel"

[Hal05] Halevi, *A plausible approach to computer-aided cryptographic proofs*

[3] We use the terms *cryptographic building blocks* and *cryptographic primitives* interchangeably in this thesis.

[DY83] Dolev and Yao, "On the Security of Public Key Protocols"

symbolic model facilitates automated analysis, as implemented for example in the tools ProVerif [Bla16] and Tamarin [Mei+13]. However, cryptographic building blocks are not perfect in reality, there is always at least a small possibility that the adversary guesses a key. The computational model considers a more realistic adversary and allows to quantify the probability with which an adversary breaks desired security properties. In this model, the adversary is described as a probabilistic Turing machine, messages are bitstrings, cryptographic building blocks are modeled as functions from bitstrings to bitstrings, and they are not perfect, but assumed to be broken with a certain negligible probability [GM84]. An analysis in this model aims to establish an upper bound for the probability that a security property of the protocol can be broken, and, more constructively, to inform protocol parameter choices like key sizes and the number of users to keep this probability negligible in a real-world scenario. Proofs in the computational model are reductionist: they reduce the security of a cryptographic system to hardness assumptions on cryptographic building blocks used within it. The style of cryptographic proofs in the computational model has evolved over the years, and nowadays, *code-based* game-playing proofs have become the norm for the type of protocols considered in this thesis [BR04].

[Bla16] Blanchet, "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif"

[Mei+13] Meier et al., "The TAMARIN Prover for the Symbolic Analysis of Security Protocols"

[GM84] Goldwasser and Micali, "Probabilistic Encryption"

[BR04] Bellare and Rogaway, *Code-Based Game-Playing Proofs and the Security of Triple Encryption*

RELATED WORK. In this thesis, we use CryptoVerif, a proof assistant working in the computational model and specialized on game-based proofs for protocols and less on cryptographic building blocks. It provides a relatively high degree of automation which allows to treat complex protocols. Various automated proof assistants have been developed to formalize game-based proofs. CryptoVerif is not foundational in that its entire implementation is part of the trusted computing base: while its game transformations are proven correct on paper, the implementation itself is not formally proven correct. CertiCrypt was an attempt to create a foundational proof assistant for cryptographic proofs based on Coq [BGZB09]. The successor EasyCrypt has been developed because CertiCrypt did not scale well enough [Bar+11] – the desire to be foundational has been dropped as a compromise, and so EasyCrypt's implementation and external SMT solvers are part of its trusted computing base. Still, EasyCrypt requires users to indicate all intermediate games and write proofs of indistinguishability manually, the tool only checks them. This makes EasyCrypt struggle for large protocols like TLS and Signal, that can be treated in CryptoVerif [BBK17; KBB17]. However, EasyCrypt works better for cryptographic building blocks [Alm+19] because its input language is more expressive. In contrast to CryptoVerif, e. g., it can handle loops with mutable state, and allows to organise model code in modules. Furthermore, it can express generic reduction proofs and in particular generic hybrid arguments. These reasons make EasyCrypt better equipped for use with other kinds of security notions than only game-based ones. First, it has been used for simulation-based security, that is usually employed for secure multi-party computation protocols [Haa+18]. Second, there is a development under the name of EasyUC [CSV19], using EasyCrypt for the universal composability (UC) framework [Can00]. The UC model provides strong composability theorems. However, proofs are hard to obtain for efficient protocols, or, necessitate changes to the protocol. An interesting

[BGZB09] Barthe et al., "Formal Certification of Code-Based Cryptographic Proofs"

[Bar+11] Barthe et al., "Computer-Aided Security Proofs for the Working Cryptographer"

[BBK17] Bhargavan et al., "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate"

[KBB17] Kobeissi et al., "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach"

[Alm+19] Almeida et al., "Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3"

[Haa+18] Haagh et al., "Computer-Aided Proofs for Multiparty Computation with Active Security"

[CSV19] Canetti et al., "EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security"

[Can00] Canetti, *Universally Composable Security: A New Paradigm for Cryptographic Protocols*

area of real-world application of UC is password-based authenticated key exchange, e. g., a recent paper by Gu, Jarecki, and Krawczyk [GJK21].

[GJK21] Gu et al., "KHAPE: Asymmetric PAKE from Key-Hiding Key Exchange"

The Foundational Cryptography Framework [PM15] is built based on the Coq proof assistant, and thus foundational as in that is has a small trusted computing base that can be reviewed manually. It can be used to do exact security proofs, and has been used for cryptographic building blocks [Ber+15] and not for complex protocols; it is constrained by the same scaling problems as CertiCrypt and EasyCrypt.

[PM15] Petcher and Morrisett, "The Foundational Cryptography Framework"

[Ber+15] Beringer et al., "Verified Correctness and Security of OpenSSL HMAC"

Another possibility to obtain cryptographic guarantees in the computational model is to work in the symbolic model and use *computational soundness* results to lift analyses to the computational model. Then, one can benefit from the high degree of automation facilitated by the symbolic model. A survey by Cortier, Kremer, and Warinschi gives an overview of the state of computational soundness literature in the year 2011 [CKW11], summarizing computational soundness results and under which conditions they hold. The authors also survey computationally sound proof systems that "use symbolic methods and techniques in computational models". A more recent computationally sound proof system has been introduced in 2014 by Bana and Comon, for a bounded number of sessions [BC14]. Intuitively, compared to symbolic models presented above, such models define what the adversary *cannot* do (up to a negligible probability). An extension to an arbitrary number of sessions, and the development of an interactive tool working in this framework, called Squirrel, have been presented by Baelde, Delaune, Jacomme, Koutsos, and Moreau [Bae+21]. A current practical limitation of this methodology is that the addition of new axioms to the prover requires extending the tool's OCaml code – it is not yet possible to define axioms using the input language, like in CryptoVerif and EasyCrypt.

[CKW11] Cortier et al., "A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems"

[BC14] Bana and Comon-Lundh, "A Computationally Complete Symbolic Attacker for Equivalence Properties"

[Bae+21] Baelde et al., "An Interactive Prover for Protocol Verification in the Computational Model"

Back to the computational model, a recently introduced and promising methodology for game-based security proofs that provides some composition theorems is that of state-separating proofs [Brz+18]. A recent not-yet-published work applies it to parts of the Messaging Layer Security (MLS) secure group messaging protocol [BCK21]. A first mechanization in Coq has been done with SSProve [Aba+21], and work is being done for implementation in EasyCrypt [DKO21].

[Brz+18] Brzuska et al., "State Separation for Code-Based Game-Playing Proofs"

[BCK21] Brzuska et al., *Cryptographic Security of the MLS RFC, Draft 11*

[Aba+21] Abate et al., "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq"

[DKO21] Dupressoir et al., *Bringing State-Separating Proofs to EasyCrypt - A Security Proof for Cryptobox*

## 1.1 Outline and Contributions

In Chapter 2, we introduce the proof methodology of game-based proofs, and the CryptoVerif proof assistant based on the example of the Needham-Schroeder-Lowe authentication protocol.

In the first part of this thesis, we present work supporting the main hypothesis of this thesis: that cryptographic proof assistants, and in particular CryptoVerif are practical for writing cryptographic proofs for complex real-world protocols using highly-detailed models, and that such analysis can sustantially inform the design of cryptographic protocols.

### 1.1.1 *WireGuard*

In Chapter 3, we present cryptographic proofs of a comprehensive list of security properties for the WireGuard Virtual Private Network (VPN) protocol. The WireGuard VPN has quickly seen widespread adoption, because of its modern and simple design and implementation. This makes it an important protocol worth of careful cryptographic analysis. WireGuard's success continues: In the year 2020, one year after our cryptographic analysis was published, it has been integrated into the Linux kernel. A hand-written cryptographic proof of the WireGuard protocol was published before our work, in a paper by Dowling and Paterson [DP18]. They modify the protocol slightly to make it amenable for analysis as authenticated key exchange protocol in an eCK model. We analyse the entire WireGuard protocol as it is, in an ACCE-like model [Jag+12] as secure authenticated channel protocol, including transport data messages that have been excluded by [DP18]. Dowling and Paterson conduct their proof assuming a cyclic group. However, the elliptic curve employed by WireGuard, Curve25519, does not constitute a cyclic group. For our analysis, we create a detailed model of this elliptic curve, taking into account its particularities like, for example, that there are equivalent public keys. Overall, our analysis confirms WireGuard's cryptographic security and uncovers an attack of theoretical interest, that underlines the importance of detailed modelling of non-prime-order elliptic curves.

[DP18] Dowling and Paterson, "A Cryptographic Analysis of the WireGuard Protocol"

[Jag+12] Jager et al., "On the Security of TLS-DHE in the Standard Model"

### 1.1.2 *Hybrid Public Key Encryption*

Chapter 4 and Chapter 5 are concerned with the Hybrid Public Key Encryption (HPKE) standard that has been published as RFC 9180 in February 2022. We shaped its development substantially and became an official co-author. In Chapter 4, we present a detailed analysis of one of HPKE's modes, indicating and discussing exact security bounds, in the spirit of practice-oriented provable security. Continuing our effort of creating detailed models of elliptic curves, we introduce the framework of *nominal groups* to capture prime-order and non-prime-order groups in one single model. In Section 5.1, we present an analysis we did beforehand on all of HPKE's modes. In Section 5.2, we summarize the most important contributions that we made to the cryptographic design of the standard. In Section 5.2.5, we describe how we use an implementation of HPKE developed by us to produce interesting information that contributed to the standard. During the work on the HPKE standard, a particular advantage of computer-aided proofs became practically useful: the models and proofs were easily adaptable to new drafts of the standard, with the peace-of-mind that the proof assistant checks everything again. This would be much more tedious and error-prone to do entirely by hand.

### 1.1.3 *cv2fstar: Linking Computational Proofs and Implementations*

In the second part of the thesis, we consider the problem of obtaining cryptographic guarantees on executable code. This aims to close a gap that is left open most of the time: both cryptographic proofs and implementations are usually produced starting from a specification; however, there is no assurance

that both interpretations match, and thus, that the cryptographic guarantees proved for the specification actually hold for the implementation. In the end, it is the implementation of a cryptosystem that matters for practical security, and so it is an important real-world problem to tackle. We chose the approach of linking the CryptoVerif proof assistant with the F* programming language and proof assistant. To this end, in Chapter 6, we present cv2fstar, a tool that translates CryptoVerif models to executable specifications in F*. This follows a long line of related work that we summarize in Section 6.2. In short, a link to executable specifications in another proof assistant, from highly-detailed models of complex protocols, written in a proof assistant like CryptoVerif that can prove concrete cryptographic security bounds, has not been done previously. Our case study on the Needham-Schroeder-Lowe protocol shows that our CryptoVerif models can be so detailed that we do not need to fill many gaps when linking to executable code. Notably, gaps that the CryptoVerif models hide using non-cryptographic assumptions like message encoding functions, can be closed by cv2fstar: the compiler generates F* lemmas as proof obligations for these assumptions, such that they can be proved for the actual implementation. In a more global view, cv2fstar connects CryptoVerif to the large F* ecosystem including the HACL* verified cryptographic library.

Finally, in Chapter 7, we conclude, share our thoughts – looking back on the work presented in this thesis – on the placement of mechanized proofs within the cryptographic community, and lay out avenues for future work.

### 1.1.4 *Dependencies between Chapters*

Readers not familiar with game-based proofs or the CryptoVerif proof assistant are advised to read Chapter 2 before the remainder of the thesis. Chapter 3 and Chapter 4 can be read independently from the other chapters. Chapter 5 benefits from the introduction to HPKE and builds upon the formalisation of HPKE presented in Chapter 4, so we advise to read at least Section 4.1 and Section 4.6 before Chapter 5. Chapter 6 can be read independently, although it is advised to read Section 2.5 before, to better understand the case study presented in Section 6.15.

## 1.2 List of Publications

During my time as PhD candidate at Inria, I was involved in the following conference publications, standards, and preprints.

### Conference Publications

Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. “A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol”. In: *4th IEEE European Symposium on Security and Privacy*. Full version: https://hal.inria.fr/hal-02100345. Stockholm, Sweden: IEEE Computer Society, June 2019, pp. 231–246. DOI: 10.1109/EuroSP.2019.00026. See Chapter 3.

Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. “Analysing the HPKE Standard”. In: *EU-*

*ROCRYPT 2021, Part I*. ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. LNCS. Springer, Heidelberg, Oct. 2021, pp. 87–116. DOI: 10.1007/978-3-030-77870-5_4, Full version: https://eprint.iacr.org/2020/1499. See Chapter 4.

REQUEST FOR COMMENTS (RFC)

Richard L. Barnes, Karthik Bhargavan, Benjamin Lipp, and Christopher A. Wood. *Hybrid Public Key Encryption*. RFC 9180. RFC Editor, Feb. 2022, pp. 1–107. URL: https://www.rfc-editor.org/rfc/rfc9180.html. See Section 5.2.

Formally, HPKE is an *Informational* Request for Comments (RFC) published by the IRTF, and only *Standards-Track* RFCs by the IETF can become official *Internet Standards*. Informally, RFCs like HPKE are also called a "standard" within the community. Thus, we use the terms *RFC* and *standard* interchangeably for HPKE, in this thesis. RFC 7418 [4], and RFC 2014 [5] provide an overview of how the IRTF and its research groups work. The Crypto Forum Research Group (CFRG) is one of them. [6]

[4] https://www.rfc-editor.org/rfc/rfc7418.html

[5] https://www.rfc-editor.org/rfc/rfc2014.html

[6] https://irtf.org/cfrg

PREPRINT

Benjamin Lipp. *An Analysis of Hybrid Public Key Encryption*. Cryptology ePrint Archive, Report 2020/243. https://eprint.iacr.org/2020/243. 2020. See Section 5.1.

# 2

# *The CryptoVerif Proof Assistant*

CryptoVerif is a proof assistant that formalizes cryptographic proofs by sequences of games, also called game hopping. Before introducing CryptoVerif, we briefly recap this proof methodology.

## 2.1 Proofs Based on Sequences of Games

Game-based security notions model security properties by a game that gives an adversary a challenge to solve – solving the challenge is then interpreted as breaking the security property. We further describe this approach based on the tutorial paper by Victor Shoup [Sho04], and with references to an excerpt [FM21] from the hash book by Mittelbach and Fischlin [MF21] and to the introduction of code-based proofs by Bellare and Rogaway [BR04]. The game exposes oracles to communicate with the adversary, and the oracles might expect some input from the adversary, and might return an output. Shoup begins by saying that the game can be modeled as a probability space because game and adversary are probabilistic processes: oracles perform a possibly probabilistic computation, and their outputs are the random variables that the adversary sees from the probability space. The adversary is equally performing probabilistic computation. We assume that all computation is efficient, i. e., in bounded polynomial time. [1]

[Sho04] Shoup, *Sequences of games: a tool for taming complexity in security proofs*

[FM21] Fischlin and Mittelbach, *An Overview of the Hybrid Argument*

[MF21] Mittelbach and Fischlin, *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography*

[BR04] Bellare and Rogaway, *Code-Based Game-Playing Proofs and the Security of Triple Encryption*

[1] In this thesis, we do not consider quantum computation nor a quantum adversary.

Adversary Advantage. We separate two kinds of challenges that the adversary can face: *computational tasks*, where the adversary has to compute some value that satisfies some condition, e. g., forge a ciphertext, and *distinguishing tasks*, where the adversary has to decide between several but usually two options, e. g., determine which one of two messages the challenge oracle encrypted, or determine if the adversary is interacting with the left or right version of a game (or the "real" or "ideal" version, or the "real" or "random" version). We call the probability that the adversary solves the challenge, i. e., breaks the security property, the adversary *advantage* against the challenge. For computational tasks, this is simply the probability that the adversary succeeds to compute a value satisfying the winning condition. For distinguishing tasks between two options, we are interested in how much better the adversary is than guessing: here, we define the advantage as the difference between the probability that the adversary makes the right choice and the probability that uniform guessing wins (which is $1/2$ when naively

guessing a secret bit).

ASYMPTOTIC AND CONCRETE SECURITY. There are two frameworks for the evaluation of the adversary advantage: asymptotic security and concrete security (sometimes called exact security). In the framework of *asymptotic security*, the adversary advantage is evaluated only qualitatively. In this framework, the game is parametrized by a *security parameter* $\lambda \in \mathbb{N}$. Parameters of the game like key sizes and the number of calls that the adversary is allowed to issue to oracles, as well as the length of oracle inputs and outputs are assumed to be polynomial in $\lambda$. Security properties are defined to hold if the adversary advantage is *negligible* in $\lambda$: a function $\epsilon(\lambda)$ is negligible if for all polynomials $p$, there exists $\lambda_0 \in \mathbb{N}$ such that for all $\lambda \geq \lambda_0, \epsilon(\lambda) \leq \frac{1}{p(\lambda)}$ [FM21]. A proof that tries to establish that the adversary advantage against the security property of some system is negligible usually assumes that some underlying cryptographic building blocks are secure – specifically, we assume that the adversary advantage against these building blocks is negligible. A result in the asymptotic framework means that there exists a $\lambda$ for which the security property holds, however it does not give an indication on how to select the actual parameters, like key sizes, such that the system is secure. We use this framework in our analysis of WireGuard, see Chapter 3.

[FM21] Fischlin and Mittelbach, *An Overview of the Hybrid Argument*

In the framework of *concrete security*, the adversary advantage is evaluated quantitatively: theorems indicate the advantage as a function depending on the adversary advantage of breaking the cryptographic building blocks used in the system, the runtime of the adversary, the number of calls issued to oracles, and the length of oracle inputs and outputs. This framework allows to assess the security of a system for specific parameters. We use it in our analysis of HPKE's authenticated mode, see Chapter 4, where we evaluate the scheme's security level in bits.

In the following, we need a definition of *computational indistinguishability*. We adapt the definition of [FM21] to the concrete and asymptotic security framework:

**Definition 2.1** (Computational Indistinguishability ([FM21]))**.** In the concrete security framework, the advantage of a bounded-time probabilistic adversary $\mathcal{A}$ against the computational indistinguishability of two random variables $X, Y$ is

$$\mathsf{Adv}^{\mathrm{indist}}_{\mathcal{A},X,Y} = |\Pr[\mathcal{A}(X) = 1] - \Pr[\mathcal{A}(Y) = 1]| \ .$$

In the asymptotic security framework, two random variables $X, Y$ are *computationally indistinguishable* if for all probabilistic polynomial adversaries $\mathcal{A}$, the advantage

$$\mathsf{Adv}^{\mathrm{indist}}_{\mathcal{A},X,Y}(\lambda) = |\Pr[\mathcal{A}(\lambda, X(\lambda)) = 1] - \Pr[\mathcal{A}(\lambda, Y(\lambda)) = 1]|$$

is negligible.

For distinguishing tasks, the proof goal is to evaluate the adversary advantage against the computational indistinguishability of the left and the right version of the game (in the concrete security framework), or to

show that the two versions are computationally indistinguishable (in the asymptotic security framework).

For computational tasks, the proof goal is to evaluate the adversary advantage to compute a value satisfying the winning condition (in the concrete security framework), or to show that this advantage is negligible (in the asymptotic security framework).

The de facto standard technique to write proofs for game-based security notions consists in formulating a sequence of games, sometimes called game hopping. Starting from an initial game, transformations are gradually made to the game, producing a sequence of games. For distinguishing tasks, the sequence stops at the other version of the game (the right, the ideal, or the random version of the game). For computational tasks, the sequence stops at a game for which it is straightforward to establish that the adversary advantage is negligible, or for which it is straightforward to quantify the adversary advantage in the concrete security framework. In principle, one could reason directly about the adversary advantage on the initial game. The interest of formulating a sequence of games is to reduce the complexity of the proof, to make it easier to understand.

In the asymptotic security framework, if any two consecutive games in the sequence are computationally indistinguishable, then the initial and the final game are computationally indistinguishable and the security property holds [FM21]. In the concrete security framework, for each two consecutive games, a bound is specified for the adversary advantage to computationally distinguish the distribution of the two game's outputs. An overall advantage can be computed with which the adversary computationally distinguishes the initial and the final game [BR04].

[FM21] Fischlin and Mittelbach, *An Overview of the Hybrid Argument*

[BR04] Bellare and Rogaway, *Code-Based Game-Playing Proofs and the Security of Triple Encryption*

Shoup [Sho04] describes three types of game transformations that we introduce briefly in the following.

[Sho04] Shoup, *Sequences of games: a tool for taming complexity in security proofs*

TRANSFORMATIONS BASED ON INDISTINGUISHABILITY. Transformations of this type from a game $G_0$ to a game $G_1$ rely on the assumption that some other two distributions are computationally indistinguishable. Thus, the proof that the distributions of $G_0$ and $G_1$ are indistinguishable is reduced to this assumption, in a *proof by reduction*: If an adversary can distinguish $G_0$ and $G_1$, this would mean that the adversary could also distinguish the other two distributions that are assumed to be indistinguishable. The adversary advantage to distinguish $G_0$ and $G_1$ has then a bound by the adversary advantage to break the assumption (in the exact security framework), or is negligible (in the asymptotic security framework).

An example for this type of transition is using a Decisional Diffie-Hellman (DDH) assumption to replace a Diffie-Hellman shared secret $g^{ab}$ in a game $G_0$ by a random group element in $G_1$. If the adversary can distinguish $G_0$ and $G_1$, it could be used to construct an adversary against DDH.

TRANSFORMATIONS BASED ON FAILURE EVENTS. Here, a game is transformed into a game that has the same output distribution except if a failure event $F$ occurs. The adversary advantage to distinguish the two games is the probability of the failure event $F$. This is backed by the following lemma.

**Definition 2.2** (Difference Lemma [Sho04])**.** Let $A, B, F$ be events defined

in some probability distribution, and suppose that $A \wedge \neg F \Leftrightarrow B \wedge \neg F$. Then $|\Pr[A] - \Pr[B]| \leq \Pr[F]$.

For an example of this type of transformation, consider a game $G_0$ where an AEAD scheme with nonces is used. Many AEAD schemes with nonces lose at least some security properties in case a nonce is reused. Consider a second game $G_1$ that is the same as $G_0$ but it does not encrypt if a nonce is reused and instead aborts with a failure event. The adversary advantage to distinguish the games is thus the probability that a nonce is reused.

BRIDGING STEPS. These transformations do not change the output distributions, but serve only to prepare for transformations of the other two types. They do so by changing computations in an equivalent way, for example by reordering independent computations, deleting unnecessary computations, or inlining definitions.

## 2.2 INTRODUCTION TO CRYPTOVERIF

CryptoVerif is a proof assistant that helps to write game-based proofs. It formalizes the proof methodology by sequences of games presented above, and can be used for analyses in the asymptotic and the concrete security framework. Games are expressed in a well-specified language [Bla17], inspired mainly by the applied pi-calculus [AF01], using a probabilistic semantics instead of a non-deterministic one. CryptoVerif provides a high degree of automation. The user is required to specify the initial game, the proof goals, and optionally the game transformation steps. The transformation steps are indicated with concise commands that instruct CryptoVerif to create the sequence of games itself, writing the intermediary games automatically.

[Bla17] Blanchet, *CryptoVerif: A Computationally-Sound Security Protocol Verifier*

[AF01] Abadi and Fournet, "Mobile Values, New Names, and Secure Communication"

CryptoVerif can use all three types of transformations described in the previous section, and all transformations applied by CryptoVerif are guaranteed to produce a sequence of computationally indistinguishable games. CryptoVerif has an automatic mode where it can find which transformations to apply based on a built-in proof strategy. This way, simple protocols can be proven completely automatically, without indicating any transformation steps. This is different from EasyCrypt [Bar+11] where the user has to manually write all intermediary games and the proofs that they are computationally indistinguishable.

[Bar+11] Barthe et al., "Computer-Aided Security Proofs for the Working Cryptographer"

CryptoVerif can prove secrecy, authentication, and indistinguishability properties. Secrecy corresponds to real-or-random indistinguishability where CryptoVerif proves that a variable in the game is indistinguishable from a random value of the same type. Authentication properties are proved by correspondences between events that are issued in the game. For example, a correspondence could express that if some event has been executed, then some other event has been executed before, where events may represent that a participant $A$ thinks they talk to a participant $B$, and conversely. For indistinguishability properties, the user specifies two games, and CryptoVerif transforms both of them until it can prove that they are indistinguishable. We discuss these three kinds of properties in more detail in Section 2.4.

For each transformation, CryptoVerif keeps track of the adversary advantage to distinguish the two games, and computes the final advantage based on them. A successful proof and a final advantage are reached when in a game, all desired properties can be proved. This is then the final game. As all transformations have an advantage of being detected negligible in the security parameter, the same holds for the final advantage. This means that when CryptoVerif concludes with a proof, an asymptotic proof for the security properties has been found. Furthermore, CryptoVerif indicates the exact security bound depending on the number of oracle queries, the length of inputs, the size of types, the execution time of the adversary, and the adversary's advantage to break the security properties of cryptographic building blocks.

CryptoVerif comes with a pre-defined set of cryptographic assumptions, and users can also specify their own using CryptoVerif's input language. There are limitations to the game transformations that can be used with CryptoVerif. For example, cryptographic assumptions have to be expressed in multi-instance versions, which means that a proof reducing multi-user security to two-user security cannot be formalized in CryptoVerif. Furthermore, CryptoVerif cannot express generic hybrid arguments as described in [FM21]. However, this and other simplifications allow the high degree of automation in CryptoVerif, that makes the tool fit to handle large real-world protocols like TLS [BBK17] and WireGuard, see Chapter 3.

In the following, we describe CryptoVerif's syntax and semantics.

[FM21] Fischlin and Mittelbach, *An Overview of the Hybrid Argument*

[BBK17] Bhargavan et al., "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate"

## 2.3 Syntax and Semantics: Overview

CryptoVerif uses a language mainly inspired by the applied pi-calculus to represent games [AF01]. This is a process calculus with probabilistic semantics, developed for the purpose of analysing cryptographic protocols [Bla17]. Process calculi are fitted for the use case of interactive protocols because they permit to model their concurrent nature with parallel executions.

[AF01] Abadi and Fournet, "Mobile Values, New Names, and Secure Communication"

[Bla17] Blanchet, *CryptoVerif: A Computationally-Sound Security Protocol Verifier*

CryptoVerif has two syntax frontends that are functionally equivalent: the channels frontend that is adapted for users familiar with process calculi, and the oracles frontend that is adapted for users more familiar with game-based proofs from cryptographic literature.

CryptoVerif models the adversary as an evaluation context of the game. [Bla17] shows that any bounded-time probabilistic Turing machine that communicates with the game via oracles, can be expressed as an evaluation context.

In the following, we use the oracles frontend to introduce CryptoVerif's syntax and semantics. Figure 2.1 gives an overview of the most important syntax elements, and we cover them briefly in this and the following sections.

Parallel execution is modeled via parallel composition of possibly different oracles, for example $Q|Q'$, and replication of one oracle, that is foreach $i \leq n$ do $Q$, which intuitively corresponds to an $n$ times parallel composition of $Q$. The value $i$ is called a *replication index*. Interaction with the adversary is modeled via oracle input parameters and oracle return values. At the beginning of the game and after each oracle return, the control

| | |
|---|---|
| $M, N ::=$ | terms |
| $i$ | replication index |
| $x[M_1, \ldots, M_m]$ | variable access |
| $f(M_1, \ldots, M_m)$ | function application |
| $x \stackrel{R}{\leftarrow} T; N$ | random number |
| let $p = M$ in $N$ else $N'$ | assignment (pattern-matching) |
| let $x : T = M$ in $N$ | assignment |
| if defined$(M_1, \ldots, M_l) \wedge M$ then $N$ else $N'$ | conditional |
| find[unique?] $(\bigoplus_{j=1}^{m} u_{j1}[\tilde{i}] = i_{j1} \leq n_{j1}, \ldots, u_{jm_j}[\tilde{i}] = i_{jm_j} \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M'_j$ then $N_j)$ else $N'$ | array lookup |
| insert $\mathrm{tbl}(M_1, \ldots, M_l); N$ | insert in table |
| get[unique?] $\mathrm{tbl}(p_1, \ldots, p_l)$ suchthat $M$ in $N$ else $N'$ | get from table |
| event $e(M_1, \ldots, M_l); N$ | event |
| event_abort $e$ | event $e$ and abort |
| $p ::=$ | pattern |
| $x : T$ | variable assignment |
| $f(p_1, \ldots, p_m)$ | function application |
| $=M$ | comparison with a term |
| $Q ::=$ | oracle definitions (input process) |
| 0 | nil |
| $Q \mid Q'$ | parallel composition |
| foreach $i \leq n$ do $Q$ | replication $n$ times |
| $O(p_1, \ldots, p_l) := P$ | oracle definition |
| $P ::=$ | oracle body (output process) |
| return$(M_1, \ldots, M_l); Q$ | oracle output |
| $x \stackrel{R}{\leftarrow} T; P$ | random number |
| let $p = M$ in $P$ else $P'$ | assignment |
| if defined$(M_1, \ldots, M_l) \wedge M$ then $P$ else $P'$ | conditional |
| find[unique?] $(\bigoplus_{j=1}^{m} u_{j1}[\tilde{i}] = i_{j1} \leq n_{j1}, \ldots, u_{jm_j}[\tilde{i}] = i_{jm_j} \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then $P_j)$ else $P$ | array lookup |
| insert $\mathrm{tbl}(M_1, \ldots, M_l); P$ | insert in table |
| get[unique?] $\mathrm{tbl}(p_1, \ldots, p_l)$ suchthat $M$ in $P$ else $P'$ | get from table |
| event $e(M_1, \ldots, M_l); P$ | event |
| event_abort $e$ | event $e$ and abort |
| yield | end |

FIGURE 2.1: Syntax of CryptoVerif's process calculus in the oracles frontend. Adapted from [Bla17]

is given to the adversary, who can decide which oracle to call next. CryptoVerif works in the computational model, and so, input and return values are bitstrings, and functions are from bitstrings to bitstrings. CryptoVerif allows to declare types that represent a certain subset of the bitstrings, and functions that work on any of the declared types. Security assumptions that express the computational indistinguishability of two processes $Q, Q'$ up to probability $p$ are noted as $Q \approx_p Q'$.

A CryptoVerif input file consists of a list of declarations followed by a structure of oracle definitions. This oracle structure describes the security game. The declarations contain type and function declarations, cryptographic and non-cryptographic assumptions, and queries that specify the security properties to prove.

## 2.4 Security Queries

Secrecy Properties. The proof goal of secrecy of a variable `k` is stated by the query `query secret k`. The meaning of this query is multi-session secrecy of `k`: CryptoVerif proves that the sequence of values of `k` over all its replications (if `k` is defined in an oracle under replication) is computationally indistinguishable from a sequence of the same length of randomly sampled values of the same type. Concretely, CryptoVerif proves this query by verifying that the values of `k` for different replication indices are independent, and that for each replication, no oracle output depends on `k`. For the formal definition, see Definition 7 in [Bla17].

[Bla17] Blanchet, *CryptoVerif: A Computationally-Sound Security Protocol Verifier*

In our analysis of $\mathsf{HPKE}_{\mathsf{Auth}}$, we use a secrecy query to prove the confidentiality security notions of HPKE displayed in Listing 4.6: a challenge bit $b$ is used by the challenge oracle to choose between two same-length plaintexts submitted by the adversary.

Authentication-Like Properties. Correspondence queries permit to prove correspondence properties between events. There are three different variants that we want to discuss here. First, queries like "if an event $A$ has been executed, then an event $B$ has been executed before". These are *non-injective* correspondences. For WireGuard, we use such a query to prove that the first protocol message cannot be forged if no key was compromised. However it can be replayed. Second, queries like "for each occurrence of the event $A$, there is a distinct occurrence of an event $B$". These are called *injective* correspondences and permit to prove authentication properties where replay is not possible. Third, queries like "if an event $A$ and an event $B$ have been executed, then some equation holds". In WireGuard, we use these to prove correctness of the protocol and resistance against unknown key-share attacks: If two parties have the same view on a protocol transcript, then they calculate the same key; if two parties calculate the same key, then they have the same view on the protocol transcript.

These three are only motivational examples. In fact, a non-injective correspondence is an implication between two logical formulae $\psi \Rightarrow \phi$, and these formulae can contain events. The grammar for logical formulae $\phi$ is the following [Bla07]:

[Bla07] Blanchet, "Computationally Sound Mechanized Proofs of Correspondence Assertions"

| | | |
|---|---|---|
| $\phi ::=$ | | formula |
| | $M$ | term |
| | event($e(M_1, , M_l)$) | event |
| | $\phi_1 \vee \phi_2$ | conjunction |
| | $\phi_1 \wedge \phi_2$ | disjunction . |

The formula $M$ holds if the term $M$ evaluates to true, the formula event($e(M_1, \ldots, M_l)$) holds if the event has been executed, and conjunction and disjunction are defined as usual. For injective correspondences, the grammar for logical formula is extended with inj-event($e(M_1, \ldots, M_l)$). The left-hand formula $\psi$ is then a conjunction of injective or non-injective events.

The algorithm to check correspondences is much more involved than the one that checks secrecy. We only present the rough idea and refer to [Bla07] for more details. For all program points in a model, CryptoVerif collects true facts, which can be the value a variable is set to, the fact that a variable is defined, and the fact that an event has been executed. For injective events, CryptoVerif also collects information on the replication indices of the events. To prove a non-injective correspondence $\psi \Rightarrow \phi$, CryptoVerif collects all facts that hold at program points of events in $\psi$ and shows that these facts imply $\phi$ using an equational prover. For injective correspondences, it is shown that if the replication indices of two executions of injective events in $\psi$ are different, then the replication indices of the corresponding executions of the considered injective event of $\phi$ are also different [Bla07].

[Bla07] Blanchet, "Computationally Sound Mechanized Proofs of Correspondence Assertions"

INDISTINGUISHABILITY BETWEEN TWO GAMES. For proofs of computational indistinguishability of two games, a CryptoVerif input file consists of a list of declarations followed by two structures of oracle definitions. To prove indistinguishability, CryptoVerif checks whether the two games are equivalent, term-by-term, while allowing variable names to be different. For example, we use this in our analysis of $\mathsf{HPKE}_{\mathsf{Auth}}$ to prove the confidentiality and authentication security notions of the KEM displayed in Listing 4.3, Listing 4.4, and Listing 4.5.

In the following, we explain the remainder of CryptoVerif's language elements based on an example.

## 2.5 EXAMPLE: THE NEEDHAM-SCHROEDER-LOWE PROTOCOL

The Needham-Schroeder and the Needham-Schroeder-Lowe (NSL) protocols have been the "Hello World" of security protocol verification methodologies since many years. We use NSL as an example to introduce CryptoVerif, and also for our first case study on cv2fstar, see Section 6.15.

The Needham-Schroeder shared-key and public-key protocols [NS78] have been presented in the year 1978, based on symmetric encryption and public-key encryption, respectively. The main goal of the protocols is to mutually authenticate two participants. In 1995, Lowe presented an attack on the public-key protocol and proposed an improved version that is called Needham-Schroeder-Lowe protocol [Low95] nowadays.

[NS78] Needham and Schroeder, "Using encryption for authentication in large networks of computers"

[Low95] Lowe, "An attack on the Needham-Schroeder public-key authentication protocol"

In the Needham-Schroeder-Lowe protocol, two participants $A$ and $B$ use a trusted key server to retrieve the other participant's public key. The

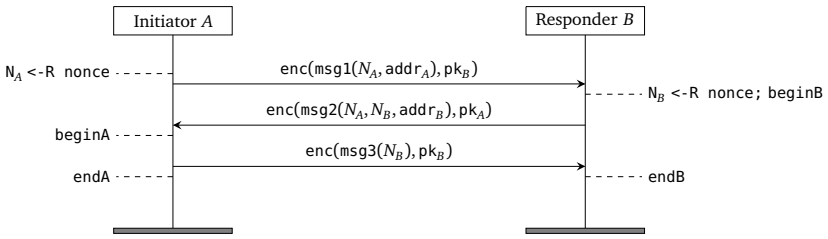



FIGURE 2.2: The simplified version of the Needham-Schroeder-Lowe protocol that we use. We assume that $A$ and $B$ possess a private key and know the other participant's public key $\mathtt{pk}_{B,A}$. We assume that they have an address $\mathtt{addr}_{A,B}$ and know the one of the other participant. The events `beginA`, `beginB`, `endA`, `endB` are used to express the authentication properties of the protocol.

key server authenticates its messages using a digital signature. In our case study, we use a simplified version of the protocol without communication between key server and participants. Instead, we implement the key server as tables in CryptoVerif, that the two participants can access. This of course constitutes a stronger assumption, but keeps our example simpler.

### 2.5.1 *Simplified Version of the Needham-Schroeder-Lowe Protocol*

The version of the Needham-Schroeder-Lowe (NSL) protocol that we use is shown in Figure 2.2. The protocol has three messages.

For the first message, the initiator $A$ samples a random nonce $\mathtt{N}_A$. The initiator encrypts the nonce along with its identity $\mathtt{addr}_A$ under the responder's public key $\mathtt{pk}_B$. Successful decryption of this first message marks the event `beginB` on the responder's side. The responder samples a random nonce $\mathtt{N}_B$ and encrypts the two nonces along with its identity $\mathtt{addr}_B$ under the initiator's public key $\mathtt{pk}_A$. Successful decryption of this second message marks the event `beginA` on the initiator's side. The initiator encrypts the nonce $\mathtt{N}_B$ under the responder's public key and sends this ciphertext as third message. This marks event `endA` on the initiator's side. Successful decryption of this third message marks the event `endB` on the responder's side, and the protocol is finished. Both participants now have a guarantee that they are talking to the participant that they think to which they are talking, protected from replay attacks.

### 2.5.2 *CryptoVerif Model for NSL*

We base the CryptoVerif model for our version of NSL on a model shipped with CryptoVerif[2]. Changes we made include: using the oracle frontend, removing the key server, and adapting the top-level oracle structure. Also, we use a public-key encryption scheme where encryption can fail.

[2]The file can be found in `examples/basic/needham-schroeder-pkcorrAuth.pcv`

The main process of the model contains four top-level oracles that we outsource in separate process definitions using a syntax feature of CryptoVerif: With the keyword `let`, we can encapsulate oracle definitions under a process name, and reuse it in the model with the keyword `run`. We use four such process definitions: the setup of the key pairs for the two honest participants $A$ and $B$, the possibility for the adversary to register key pairs for dishonest participants, and finally the initiator and responder oracles.

```
process
  (
    run setup()
  |
    run key_register()
  |
    run initiator()
  |
    run responder()
  )
```

These processes are in parallel composition, so the adversary can call the first oracle in any of them in any order. Each of these separate processes begins in the same style. First it is declared using `let`. Then, we declare a replication of the top-level oracle:

```
let setup() =
   foreach i ≤ Qsetup do
     setup(addr: address) :=
       ...
 }.
```

The parameter `Qsetup` indicates the upper limit for the number of calls to the setup oracle. It is declared by a line

```
param Qsetup.
```

### 2.5.3 *Types, Constants, Functions, and Equations*

Types, constants, and functions in a CryptoVerif model are declared without implementation, it is just assumed that they exist.

TYPES. CryptoVerif's language is strongly typed, type conversions need to be done explicitely. Arbitrary type conversion functions can be defined. CryptoVerif defines the built-in types `bitstring`, `bitstringbot`, and `bool`, where `bitstringbot` contains all bitstrings and the special failure symbol $\perp$ (`bottom` in code). A custom type $T$ is declared as follows in a CryptoVerif model:

```
type T.
```

Types correspond to sets of bitstrings or $\perp$. A type can be declared with options $o$:

```
type T [o].
```

The most common options are `fixed` or `bounded` to indicate fixed-length bitstrings, or bitstrings with a maximum length; and `large` to indicate that the number of elements of this type is large enough such that the probability that two randomly sampled elements collide is negligible. If two options are used, they are separated by a comma. For more options, we refer to the CryptoVerif manual that is available in the CryptoVerif download. [3] The options determine from which probability distribution values are randomly sampled with $\xleftarrow{R}$ (`<-R` in code).

[3] https://cryptoverif.inria.fr

CONSTANTS. A constant $c$ of type $T$ is declared as follows in a CryptoVerif model:

```
const c: T.
```

FUNCTIONS. A function $f$ with parameters of types $T_i, 1 \leq i \leq l$ and return type $T$ is declared as follows in a CryptoVerif model:

```
fun f(T_1, ..., T_n): T.
```

The optional `[data]` keyword can be added at the end. It indicates that $f$ is injective and has an efficiently computable inverse:

```
fun f(T_1, ..., T_n): T [data].
```

EQUATIONS. An equation is defined starting with the `equation` keyword, followed by a list of universally quantified variables with type indication, then a boolean term $M$ encoding the assumption, and optionally, the keyword `if` followed by a condition $M'$:

```
equation forall v_1:T_1, ..., v_l:T_l; M if M'.
```

Equations can be used to express non-cryptographic assumptions, like for example disjointness of function outputs, and assumptions about cryptographic building blocks, like for example correctness of an encryption scheme. This introduction does not cover the specification of cryptographic indistinguishability assumptions.

TYPES, AND SOME CONSTANTS, FUNCTIONS, AND EQUATIONS IN NSL. In the following, we list the main types used in the CryptoVerif model for NSL, and some constants and functions that go along with them.

We define `nonce` as a fixed, large type. For key-pair generation, we use a `keyseed` that is also defined as fixed and large type. Private keys `skey` and public keys `pkey` are declared `[bounded]`. The type `keypair` groups together a secret and public key. The type `ciphertext` is `[bounded]`. As mentioned earlier, encryption can fail, so we instantiate a CryptoVerif public-key encryption macro so that the return type of the encryption function is `ciphertext_opt`. CryptoVerif does not support option types directly, but we can model an option type as follows:

```
type ciphertext_opt [bounded].
fun ciphertext_some(input): ciphertext_opt [data].
const ciphertext_bottom: ciphertext_opt.
equation forall x: ciphertext;
  ciphertext_some(x) <> ciphertext_bottom.
```

The type for `plaintext` is `[bounded]`. We use probabilistic public-key encryption, and so we need random coins for encryption. They have the `[bounded]` type `encseed` in our model. Finally, the `[bounded]` type `address` for the participant's identities.

### 2.5.4 *Tables, and Their Use in the Setup and Registration Oracles*

In a CryptoVerif model, tables are declared using

```
table tbl(T_1, ..., T_l).
```

where tbl is the table name and $T_1$ to $T_l$ are the types of an entry's fields. In a process, entries can be added to a table by a term

```
insert tbl(M_1, ..., M_l); N
```

Listing 2.1: Code of the setup oracle.

```
1 setup(addr: address) :=
2  get[unique] all_keys(=addr, ign1, ign2) in (
3    yield
4  ) else
5    let kp(the_pkA: pkey, the_skA: skey) = keygen() in
6    insert trusted_keys(addr, the_skA, the_pkA);
7    insert all_keys(addr, the_pkA, true);
8    return(the_pkA)
```

where $M_1$ to $M_l$ must be terms with types matching the types of the table declaration. A table insert always succeeds in CryptoVerif, so there is no branching, and only one next term $M$ that is evaluated. Entries cannot be removed from tables.

A table is queried by a term

```
get tbl(p_1, ..., p_l) suchthat M in N' else N''
```

where $p_1$ to $p_l$ are patterns. We will explain patterns by example when they occur in the following. The query tries to find an entry with fields matching the patterns such that the term $M$ is `true`. If exactly one matching entry is found, the term evaluates to the result of $N'$. If more than one matching entries are found, one among them is chosen almost uniformly random, and the term evaluates to the result of $N'$. Otherwise, it evaluates to the result of $N''$. The keyword `[unique]` can be added directly after `get` to make it `get[unique]`. Then, CryptoVerif tries to prove that there is only ever at most one matching table entry for this query, up to negligible probability.

We use two tables, `trusted_keys` and `all_keys`. Each table entry contains data about a participant. The table `trusted_keys` contains addresses and key pairs of honest participants. The table `all_keys` contains addresses and the public keys of all participants, and a trust bit indicating if this participant is honest.

```
table trusted_keys(address, skey, pkey).
table all_keys(address, pkey, bool).
```

The setup oracle generates a key pair and writes it to both tables, with the trust bit set to `true`, see Listing 2.1. Before generating and inserting the key pair, the oracle uses a `get[unique]` query to test if a key for the given address exists already. Here, we use an equality test pattern to compare the first field of a table entry with the variable `addr`, by using the `=addr` syntax. We do this check in the key register oracle, as well. This means entries are only added to the tables for a given address, if no entry for this address exists, yet. Thus, CryptoVerif can prove that the get queries indeed have at most one matching entry. In case a key pair is already registered for address `addr`, the oracle ends with `yield`, which gives control back to the adversary, letting it know that the oracle terminated with an error. In Line 5, there is an instance of pattern matching. As we describe later in more detail, the `keygen` function returns a key pair. The `kp` function constructs a key pair given a public key and a private key. In the pattern matching, the key pair returned by `keygen` is split into a public key and a private key using the

inverse of the function `kp`. If this pattern matching fails, the oracle implicitly ends with `yield`.

The key register oracle writes public keys received from the adversary to the table `all_keys` with the trust bit set to `false`:

```
register (addr: address, pkX: pkey) :=
 get[unique] all_keys(=addr, ign1, ign2) in (
  yield
 ) else
  insert all_keys(addr, pkX, false);
  return ()
```

The initiator and responder oracles retrieve their own private key from the table `trusted_keys`, and the public key of their interlocutor from the table `all_keys`, see Section 2.5.8.

### 2.5.5 *Message Encoding Functions*

The message encoding functions `msg1`, `msg2`, and `msg3` return a value of type `plaintext` and are declared `[data]` because we need a parser to retrieve the message's fields on the recipient's side, respectively:

```
fun msg1(nonce, address):plaintext [data].
fun msg2(nonce, nonce, address):plaintext [data].
fun msg3(nonce):plaintext [data].
```

We assume that messages never collide, by using the following three equations:

```
equation forall z:nonce, t:nonce, u:address, y2:nonce, z2:address;
      msg2(z, t, u) <> msg1(y2, z2).
equation forall y:nonce, y2:nonce, z2:address;
      msg3(y) <> msg1(y2, z2).
equation forall z:nonce, t:nonce, u:address, y2:nonce;
      msg2(z, t, u) <> msg3(y2).
```

### 2.5.6 *Instantiating Public-Key Encryption*

The only cryptographic assumption that we use to prove secure our variant of NSL is IND-CCA2 security of the probabilistic public-key encryption scheme. The CryptoVerif library ships a macro that defines such a scheme and that we instantiate with the types `keyseed`, `pkey`, `skey`, `plaintext`, `ciphertext_opt`, and `encseed`. The instantiation is as follows, and we explain the other parameters step by step:

```
proba Penc.
proba Penccoll.

expand IND_CCA2_public_key_enc_all_args(
  keyseed, pkey, skey, plaintext, ciphertext_opt, encseed,
  skgen, skgen2, pkgen, pkgen2, enc, enc_r, enc_r2,
  dec_opt, dec_opt2, injbot, Z, Penc, Penccoll).
```

The probability function `Penc(t, N)` stands for the adversary advantage to break the IND-CCA2 property in time `t` for a key, using `N` queries to a decryption oracle. The collision probability between private or public keys that are independently generated is denoted by `Penccoll`.

The macro declares for us the functions `skgen` and `pkgen` that model generation of private and public keys. For encryption, it declares a deterministic encryption function `enc_r` that takes random coins as explicit parameter, and defines a probabilistic encryption function `enc` that samples random coins internally. Function declarations are always for deterministic functions in CryptoVerif, so `enc` has to use another language feature: With `letfun` definitions, CryptoVerif allows to reuse terms under a given name for convenience and better readability of the model; they are immediately inlined during the proof. Thus, `enc` is a `letfun` that wraps `enc_r`, passing the randomness to it:

```
fun enc_r(plaintext, pkey, encseed): ciphertext.
letfun enc(m: plaintext, pk: pkey) =
  r <-R encseed; enc_r(m, pk, r).
```

For decryption, the macro declares a decryption function `dec_opt`, and a function `injbot` that injects the type `plaintext` into the type `bitstringbot`. The decryption function `dec_opt` takes a value of type `ciphertext_opt` as parameter, to allow it to be called on the failure value `ciphertext_bottom`. Outside the macro, we define a letfun function `dec` as a wrapper that can be called with a value of type `ciphertext` directly. It injects the value into `ciphertext_opt` using the function `ciphertext_some` before calling `dec_opt`:

```
letfun dec(c: ciphertext, sk: skey) =
  dec_opt(ciphertext_some(c), sk).
```

The function names ending in `2` in the macro instantiation are not used in the initial game. They will appear in the game after CryptoVerif applies the IND-CCA2 transformation, replacing the occurrences of the function names without `2` by the function names with `2`. This is so that the transformation cannot match again and let CryptoVerif end up in an infinite loop.

Finally, the macro declares a function `Z` that models the leakage of the plaintext length by the encryption scheme. It is used in the definition of the IND-CCA2 transformation.

Outside the macro, we declare a function `kp` that takes a public and a private key, and returns a keypair. We define a letfun `keygen` wrapping this function, sampling the key seed internally.

```
fun kp(pkey, skey): keypair [data].
letfun keygen() = k <-R keyseed; kp(pkgen(k), skgen(k)).
```

### 2.5.7 *Events and Queries*

In a CryptoVerif model, an event can be declared by

```
event e(T_1, ..., T_l).
```

where `e` is the name of the event, and $T_1$ to $T_l$ are the types of its parameters. An event is invoked by a term

```
event e(M_1, ..., M_l); N
```

where `e` is the name of an event that has been declared in the model, and $M_1$ to $M_l$ are terms with types matching the event's declaration. An event cannot fail, so there is no branching, and the return value of the term is

the evaluation of the following term $N$. An event can also be invoked by `event_abort e` in which case the game is stopped after executing the event, and so there is no following term. CryptoVerif will try to prove that the probability that this event is invoked is negligible. This is for example used to prove that nonces are not reused in an AEAD scheme with nonces.

In our NSL model, the four events `beginA`, `beginB`, `endA`, and `endB` are all declared to take both participants addresses and the two nonces as input:

```
event beginA(address, address, nonce, nonce).
event endA(address, address, nonce, nonce).
event beginB(address, address, nonce, nonce).
event endB(address, address, nonce, nonce).
```

This way, at the appropriate stages of the protocol, the participants record what participant they believe to talk to, and based on what nonces.

As a reminder to what was presented when describing Figure 2.2, event `beginB` is issued when the responder successfully decrypted the first protocol message, event `beginA` is issued when the initiator successfully decrypted the second protocol message, event `endA` is issued after the initiator sent the third protocol message, and event `endB` is issued after the responder successfully decrypted the third protocol message. However, the events `endA` and `endB` are only issued if their interlocutor is marked as trusted in the table `all_keys`, respectively. This is because only in this situation, if two honest participants communicate, we want to prove mutual authentication. We ask CryptoVerif to prove the following correspondance queries, and it succeeds to do so in automatic mode:

```
query x:address, y:address, na:nonce, nb:nonce;
      inj-event(endA(x,y,na,nb)) ⟹ inj-event(beginB(x,y,na,nb)).
query x:address, y:address, na:nonce, nb:nonce;
      inj-event(endB(x,y,na,nb)) ⟹ inj-event(beginA(x,y,na,nb)).
```

This means that for two honest participants, for each event `endA`, there is a unique event `beginB` with the same parameters, and likewise, for each event `endB`, there is a unique event `beginA` with the same parameters. Intuitively, this ensures that if an honest participant $P_1$ believes to have concluded the protocol with another honest participant $P_2$ resulting in two nonces, then this other participant $P_2$ has actually engaged in the protocol, too, believing to talk to $P_1$, with the same nonces. Also, it ensures that there is no replay attack.

### 2.5.8 *Honest Participant Oracles*

The initiator process exposes three oracles that can be called only in sequence. This is modeled by defining the following oracles after the return of the previous oracle, respectively. The first oracle sends the first protocol message, the second oracle receives the second protocol message and sends the third protocol message, and the third oracle issues the event `endA` if the responder is honest. The responder process exposes two oracles that can be called only in sequence. The first oracle receives the first and sends the second protocol message, the second oracle receives the third protocol message and issues the event `endB` if the initiator is honest. We only cover the code of the initiator in this section, the code of the responder is included in the full `nsl.ocv` file in Appendix A.1.

Listing 2.2: Code of the oracle sending the first protocol message.

```
initiator_send_msg_1(addrA: address, addrX: address) :=
 (* the gets fail if addrA or addrX have not been
    setup by the adversary. *)
 get[unique] trusted_keys(=addrA, skA, pkA) in
 get[unique] all_keys(=addrX, pkX, trustX) in
 (* Prepare Message 1 *)
 Na <-R nonce;
 let cc1 = enc(msg1(Na, addrA), pkX) in
 let ciphertext_some(c1: ciphertext) = cc1 in
 return (c1);
```

The initiator's first oracle, `initiator_send_msg_1`, takes two addresses as parameters, see Listing 2.2. The first one instructs the oracle which identity to use as initiator, the second one is used for the responder. Thus, the adversary can request a communication between any two participants that have previously been set up either using the setup or the key register oracle. The oracle starts by trying to retrieve the private and public key for the initiator identity from the table `trusted_keys`, and the public key and trust bit for the responder identity from table `all_keys`. If any of these get requests fail, the oracle implicitly yields. If the requests are successful, a nonce is sampled, and the first protocol message is prepared and encrypted. Only if the encryption succeeds, tested by a pattern matching, the oracle returns successfully and sends the ciphertext to the adversary. If the pattern matching fails, the oracle implicitly yields.

The second oracle consumes the second protocol message and sends the third protocol messages. It takes only the ciphertext of the supposed second protocol message as parameter. Other variables, like `Na`, `addrX`, and `skA`, are still in scope from the previous oracle, because the oracles are defined in sequence.

```
initiator_send_msg_3 (c: ciphertext) :=
 let injbot(msg2(=Na, Nb, =addrX)) = dec(c, skA) in
 event beginA(addrA, addrX, Na, Nb);
 let ciphertext_some(c3) = enc(msg3(Nb), pkX) in
 return (c3);
```

The ciphertext is decrypted and if it is a valid plaintext, parsed as second message via the inverse of `msg2`, to retrieve the nonce `Nb`. Here, we use all three types of pattern matching shown in Figure 2.1: variable assignment, function application, and comparison with a term. Afterwards, event `beginA` is issued and the third message sent, if the encryption is successful.

The third oracle does not take parameters, its interest is to invoke the event `endA` if the other participant is trusted: we only want to prove authentication between trusted participants.

```
initiator_finish () :=
 if (trustX) then
 event endA(addrA, addrX, Na, Nb);
 return ()
```

The `else` branch of the condition is an implicit `yield`.

### 2.5.9 *Elements Not Covered by the Example*

ARRAYS. An extension of particular interest of CryptoVerif's calculus over other process calculi is the automatic accessibility of variable values via arrays: The value of a variable $x$ in a specific replication of an oracle can be accessed via array indices $x[\widetilde{i}]$, where $\widetilde{i}$ is the sequence of all replication indices needed to uniquely identify an oracle replication that is possibly nested. If an oracle is under multiple nested replications, the order of the replication indices is from the innermost to the outermost replication.

This is used in the `find` array lookup, which is useful for case distinctions in proofs. An array lookup `find` $i$ `<=` $n$ `suchthat defined(`$x[i]$`)` $\wedge$ $M$ `then` $N$ `else` $N'$ looks for a replication index $i$ in $n$ such that $x[i]$ is defined and the condition $M$ holds. If it finds one, it evaluates to $N$ with that index set; otherwise, it evaluates to $N'$. This can be extended to several branches by using `orfind`. The keyword `[unique]` can be used to specify that the lookup shall only succeed if exactly one index $i$ satisfying the condition is found. As a particular case of `find`, the built-in function `defined` can be used in an `if` conditional, too. An example for the usage of `find` for case distinctions in proofs is described at the end of Section 3.6.

TUPLES. In a CryptoVerif model, tuples can be constructed by wrapping multiple values into parentheses, separated by commas, like in this example: `(a, b, c)`. Tuples are of type `bitstring`. They do not appear in Figure 2.1 because they are seen as special case of a function application.

NIL. The last element from Figure 2.1 that is not yet explained is `0`. It is an input process that does nothing and can be used to make explicit that no oracles are exposed to the adversary.

This concludes the description of CryptoVerif's syntax and semantics. We are not describing how proofs are done in CryptoVerif. The correspondence queries in our NSL example are proved automatically by CryptoVerif without any proof statements necessary. The proof is done by reducing the security of the authentication property to the IND-CCA2 security of the public-key encryption scheme. A description of a proof with proof steps is included in the chapter about WireGuard, at the end of Section 3.6. A comparison of security notions in the style known from cryptography papers and their implementation in a CryptoVerif model can be found in Appendix C.2.

## 2.6 CONTRIBUTIONS TO CRYPTOVERIF'S USABILITY

During my work with CryptoVerif, I contributed various external scripts and ideas that have improved the usability of the tool. Three of them are highlighted in the following.

### 2.6.1 *Referencing Terms in Intermediary Games*

For proof guidance, CryptoVerif supports an `insert` command that is useful for case distinctions in a proof. For example, in the WireGuard proof, we introduce a case distinction in the initiator process, to treat separately the

case where the initiator starts a session with the honest participant B, instead of a dishonest participant controlled by the adversary. We do this by inserting a conditional that tests whether the public key provided by the adversary S_X_pub is equal to the public key of the honest participant S_B_pub:

```
insert after "in(c_config_initiator\\["
  "if␣pow_k(S_X_pub)␣=␣pow_k(S_B_pub)␣then";
```

Here, in(c_config_initiator is the start of an oracle definition in the channels frontend, that we indicate as regular expression to the insert command. The first line in the code snippet means that the conditional is inserted right after the declaration of the oracle, as first line in its body. CryptoVerif duplicates the code of the oracle body into the then and the else branch of the conditional. In the remainder of the proof, we can then transform the two branches separately.

When we began working on the WireGuard analysis, the feature described above was not yet available. The insert command would not take a regular expression as parameter, but an *occurrence number*. Terms in a CryptoVerif game are consecutively numbered with an occurrence number to uniquely identify them within the game. However, this is a quite brittle method to identify terms in a proof, because small changes in the initial game or any previous proof step might change this numbering, requiring a manual adjustment of the occurrence number, by inspecting the intermediary game output of CryptoVerif. The above example looked like this:

```
insert 38 "if␣pow_k(S_X_pub)␣=␣pow_k(S_r_pub)␣then";
```

The occurrence number itself does not document how it was determined. In contrast, the statement after "in(c_config_initiator\\[" is much more self-explanatory.

To mitigate this usability limitation, I wrote a Bash script that allowed to include special comments inside the proof indications of a CryptoVerif model. The script would drive the CryptoVerif proof up to a point where such a special comment appears, use the shell command embedded in the special command to extract the desired occurrence number from CryptoVerif's intermediary game output, and use this number to continue the proof. This script, and the complexity of the WireGuard proofs, eventually inspired the new feature described above. It got integrated into CryptoVerif with version 2.01. The Bash scripts are still available online, along with some other scripts that transform CryptoVerif's game and proof output into more readable forms. [4]

[4] https://github.com/blipp/cryptoverif-helpers

### 2.6.2 *Indicating Differences Between Games*

As described previously, CryptoVerif can be used to prove computational indistinguishability of two games specified in the input file. During an unfinished proof, CryptoVerif used to only tell the user that the two games are not indistinguishable, but would not give an indication at which terms they differ. It was up to the user to closely look at the intermediary game output and manually find the differences. This showed to be tedious during the proofs of the DHKEM security notions during the work on the $\mathsf{HPKE}_{\mathsf{Auth}}$ analysis. After my suggestions, this was improved for the next version of

CryptoVerif. CryptoVerif now indicates the first term in which the games differ, making such indistinguishability proofs much more accessible.

### 2.6.3 *Autocompletion for CryptoVerif's Interactive Mode*

In CryptoVerif's interactive mode, the user has to type commands like `crypto ind_cca(enc)` to guide the proof. Unlike many modern terminal emulators, CryptoVerif's prompt does not provide autocompletion, such that the user could type `cr`, press the tab key, and have CryptoVerif complete to `crypto`. Also, CryptoVerif does not provide a command history that could be accessed by the arrow keys, as in modern terminal emulators. I built a wrapper around CryptoVerif that adds both features. [5] It provides command history across usages. For autocompletion, it works with a predefined list of keywords extracted from CryptoVerif's manual and the standard library. Unfortunately, it cannot provide context-aware completion, as the wrapper has no knowledge of the current game or proof. Building context-aware completion and command history into CryptoVerif directly could be an interesting avenue for future work on usability.

[5] https://github.com/blipp/cryptoverif-completion/

# Part II

# CRYPTOVERIF CASE STUDIES ON REAL-WORLD PROTOCOLS

# 3

# *Analysing the WireGuard VPN Protocol*

This chapter and the appendix belonging to it are based on the long version [LBB19b] of the paper "A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol" published at IEEE's EuroS&P 2019 with co-authors Bruno Blanchet and Karthikeyan Bhargavan [LBB19a].

[LBB19b] Lipp et al., *A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol*

[LBB19a] Lipp et al., "A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol"

ABSTRACT. WireGuard is a free and open source Virtual Private Network (VPN) that aims to replace IPsec and OpenVPN. It is based on a new cryptographic protocol derived from the Noise Protocol Framework. In this chapter, we present the first mechanised cryptographic proof of the protocol underlying WireGuard, using the CryptoVerif proof assistant.

We analyse the entire WireGuard protocol as it is, including transport data messages, in an ACCE-style model. We contribute proofs for correctness, message secrecy, forward secrecy, mutual authentication, session uniqueness, and resistance against key compromise impersonation, identity mis-binding, and replay attacks. We also discuss the strength of the identity hiding provided by WireGuard.

Our work also provides novel theoretical contributions that are reusable beyond WireGuard. First, we extend CryptoVerif to account for the absence of public key validation in popular Diffie-Hellman groups like Curve25519, which is used in many modern protocols including WireGuard. To our knowledge, this is the first mechanised cryptographic proof for any protocol employing such a precise model. Second, we prove several indifferentiability lemmas that are useful to simplify the proofs for sequences of key derivations.

## 3.1 INTRODUCTION

The traditional distinction between a secure intranet and the untrusted Internet is becoming less relevant as more and more enterprises host internal services on cloud-based servers distributed across multiple data centres. Sensitive data that used to travel only between physically proximate machines within secure buildings is now sent across an unknown number of network links that may be controlled by malicious entities.

To maintain the security of such *distributed intranets*, the most powerful tools at the disposal of system administrators are Virtual Private Network (VPN) protocols that set up low-level secure channels between machines, and hence can be used to transparently protect all the data exchanged between

them. Indeed, all leading cloud providers now offer VPN gateways, so that enterprises can treat cloud-based servers as if they were located within their intranet.[1]

[1] https://cloud.google.com/vpn/docs/concepts/overview, https://docs.aws.amazon.com/vpc/latest/userguide/vpn-connections.html, https://azure.microsoft.com/en-us/services/vpn-gateway/

STANDARDS VS. CUSTOM PROTOCOLS. Most popular VPN solutions are based on Internet standards like IPsec [KY05] and TLS [Res18], for several reasons. First, these protocols typically have multiple interoperable implementations that are available on all mainstream operating systems, so the VPN software can be easily built as a layer on top. Second, standards are designed to be future-proof by relying on versioning and *cryptographic agility*, so that a VPN protocol can easily move from one protocol version or cryptographic algorithm to another if (say) a weakness were found on some configuration. Third, published standards typically have been closely scrutinised by numerous interested parties, and hence are believed to be less likely to contain obvious security flaws.

[KY05] Kent and Yao, *Security Architecture for the Internet Protocol*

[Res18] Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*

Conversely, using a standard protocol also has its disadvantages. Standardisation takes time, and so a standard protocol may not use the most modern cryptographic algorithms. On the contrary, the need for interoperability and backwards compatibility often force implementations to continue support for obsolete cryptographic algorithms, leading to cryptanalytic attacks [BL16] and software flaws [Beu+15]. Over time, standards and their implementations can grow to an unmanageable size that can no longer be studied as a whole, allowing logical flaws to hide in unused corners of the protocol [Bha+14a].

[BL16] Bhargavan and Leurent, "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN"

[Beu+15] Beurdouche et al., "A Messy State of the Union: Taming the Composite State Machines of TLS"

[Bha+14a] Bhargavan et al., "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS"

Consequently, many new secure channel protocols eschew standardisation in favour of a lean design that uses only modern cryptography and supports minimal cryptographic agility. The succinctness of the protocol description aids auditability, and the lack of optional features reduces complexity. Examples of this approach are the Signal protocol [MP16] used in many secure messaging systems and the Noise protocol framework [Per18].

[MP16] Marlinspike and Perrin, *The X3DH Key Agreement Protocol*

[Per18] Perrin, *The Noise Protocol Framework*

WireGuard is a VPN protocol that adopts this design philosophy [Don17]. It implements and extends a secure channel protocol derived from the Noise framework, and it chooses a small set of modern cryptographic primitives. By making these choices, WireGuard is able to provide a high-quality VPN in a few thousand lines of code, and is currently being considered for adoption within the Linux kernel. The design of WireGuard is detailed and informally analysed in [Don17], but a protocol of such importance deserves a thorough security analysis.

[Don17] Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel"

A NEED FOR MECHANISED PROOFS. Having a succinct, well-documented description is a good basis for understanding, auditing, and implementing a custom cryptographic protocol, but in itself is no guarantee that the protocol is secure. Symbolic analysis with tools like ProVerif [Bla16] and Tamarin [Mei+13] can help find logical flaws, and WireGuard already has been analysed using Tamarin [DM18]. However, symbolic analyses do not constitute a full cryptographic proof. For example, they cannot demonstrate the absence of cryptanalytic attacks on secure channels and VPNs (e.g. [BL16].)

[Bla16] Blanchet, "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif"

[Mei+13] Meier et al., "The TAMARIN Prover for the Symbolic Analysis of Security Protocols"

[DM18] Donenfeld and Milner, *Formal Verification of the WireGuard Protocol*

Cryptographic proofs provide the highest form of formal assurance, but writing proofs by hand requires significant expertise and effort, especially if the proof is to account for the precise low-level details of a real-world protocol. And as proofs get larger, the risk of introducing proof errors becomes non-negligible. All this effort is hard to justify for a custom protocol which may change as the software evolves. For example, a manual cryptographic proof for the WireGuard protocol appears in [DP18], but this proof would need to be carefully reviewed and adapted if the WireGuard protocol were to change in any way or if a variant of WireGuard were to be proposed.

[DP18] Dowling and Paterson, "A Cryptographic Analysis of the WireGuard Protocol"

We advocate the use of mechanised provers to build cryptographic proofs, so that they can be checked for errors, and can be easily modified to accommodate different variants of the protocol. In this chapter, we rely on the CryptoVerif protocol verifier [Bla08; Bla07] to build a proof of WireGuard. CryptoVerif relies on a computational model of cryptography, and generates machine-checkable proofs by sequences of games, like those manually written by cryptographers.

[Bla08] Blanchet, "A Computationally Sound Mechanized Prover for Security Protocols"

[Bla07] Blanchet, "Computationally Sound Mechanized Proofs of Correspondence Assertions"

UNCOVERING REAL-WORLD CRYPTOGRAPHIC ASSUMPTIONS. A mechanised proof also allows the analyst to experiment with a variety of cryptographic assumptions and discover the precise set of assumptions that a protocol's security depends on.

In some cases, a protocol may require an unusual assumption about a hash function, or a stronger assumption about encryption than one may have expected, and these cases can provide a guide to implementers on what concrete cryptographic algorithms should or should not be used to instantiate the protocol. For example, in our analysis of WireGuard, we find that most of the standard properties require only standard assumptions about the underlying authenticated encryption scheme (AEAD) but identity hiding requires a stronger assumption, which is satisfied by the specific algorithms used by WireGuard, but may not be provided by other AEAD constructions.

In other cases, a protocol's use of a cryptographic primitive may motivate a new, more precise model of the primitive. Protocols like WireGuard seek to depend on a small set of primitives and reuse them in different ways. For example, WireGuard relies on the Curve25519 elliptic curve Diffie-Hellman operation for an ephemeral key exchange as well as for entity authentication. It uses Curve25519 public keys both as identities and as unique nonces to identify sessions. To verify that Curve25519 is appropriate for all these usages, and to prove the absence of attacks such as replays, identity misbinding, and key compromise impersonation, we need to account for the details of the Curve25519 group, rather than rely on a generic Diffie-Hellman assumption. Hence, we propose a new model for Curve25519 in CryptoVerif and prove WireGuard secure against this model.

CONTRIBUTIONS. We present the first mechanised proof for the cryptographic design of the WireGuard VPN, including the Noise IKpsk2 secure channel protocol it uses. Our analysis is done on WireGuard v1 as specified in [Don17]. In addition to classic key exchange security for IKpsk2, we examine the identity hiding and denial-of-service protections provided by WireGuard. We

[Don17] Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel"

conclude with a discussion of the strengths and weaknesses of WireGuard, and propose improvements that would allow for stronger security theorems.

Our work also provides contributions reusable beyond the proof of WireGuard. To the best of our knowledge, this is the first mechanised proof for any cryptographic protocol that takes into account the precise structure of the Curve25519 group. We also prove a series of indifferentiability results that allow us to simplify sequences of random oracle calls, and we made several extensions to CryptoVerif that we mention in the rest of the paper when we use them. These extensions are included in CryptoVerif as of version 2.01 available at https://cryptoverif.inria.fr/.

Our models of WireGuard are available at https://cryptoverif.inria.fr/WireGuard.

## 3.2 WIREGUARD

WireGuard [Don17] establishes a VPN tunnel between two remote hosts in order to securely encapsulate all Internet Protocol (IP) traffic between them. The main design goals of WireGuard are to be simple, fast, modern, and secure. In order to establish a tunnel, a system administrator only needs to configure the IP address and long-term public key for the remote host. With this information, WireGuard can establish a secure channel, using a protocol derived from the Noise framework, instantiated with fast, modern cryptographic primitives like Curve25519 and BLAKE2. The full WireGuard VPN is implemented in a few thousand lines of code that can run on multiple platforms, but for performance, is usually run within the operating system kernel. In particular, in the year 2020, WireGuard has been incorporated into the Linux kernel with version 5.6, [2] as an alternative to IPsec.

[Don17] Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel"

[2] https://arstechnica.com/gadgets/2020/03/wireguard-vpn-makes-it-to-1-0-0-and-into-the-next-linux-kernel/

In this section, we focus on the cryptographic design of WireGuard. We begin by describing the secure channel component, then the extensions WireGuard makes for denial-of-service and stealthy operation. We end the section by detailing the concrete cryptographic algorithms used by WireGuard and the list of informal security goals it seeks to achieve.

### 3.2.1 *Secure Channel Protocol: Noise IKpsk2*

Noise [Per18] is a framework for building two-party cryptographic protocols that are secure by construction. Using the building blocks in this framework, a designer can create a new protocol that matches a desired subset of security guarantees: mutual or optional authentication, identity hiding, forward secrecy, etc. The Noise specification also includes a list of curated predefined protocols, with an informal analysis of their message-by-message security claims. WireGuard instantiates one of these protocols, which is called IKpsk2, and extends it to provide further guarantees needed by VPNs.

[Per18] Perrin, *The Noise Protocol Framework*

The secure channel protocol is depicted in Figure 3.1a, and the cryptographic computations are detailed in Figure 3.1b, using notations similar to [Don17]. Before the protocol begins, the initiator $i$ and the responder $r$ are assumed to have exchanged their long-term *static public keys* $(S_i^{\mathrm{pub}}, S_r^{\mathrm{pub}})$. Optionally, they may have also established a *pre-shared symmetric key* $(\mathrm{psk})$; if this key is absent it is set to a key-sized bitstring of zeros.

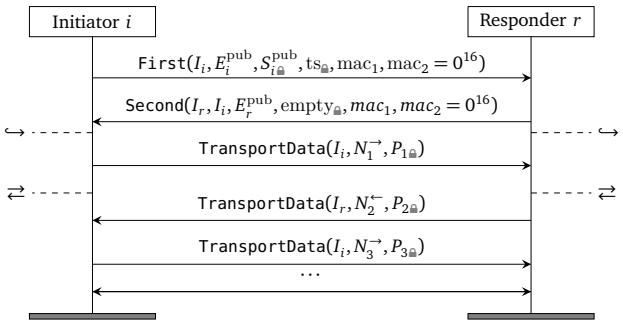
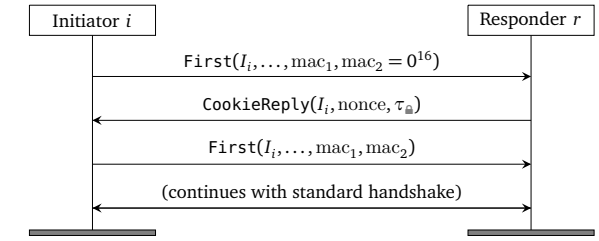



Figure 3.1a: WireGuard's protocol messages.




- - - - - - - - - CookieReply - - - - - - - - -

$R_r \leftarrow\$ \{0,1\}^{256}$ (refresh only every 2 minutes)

$\tau \leftarrow \mathsf{mac}(R_r, A_i)$ with $A_i = \mathsf{IP}_i \| \mathsf{Port}_i$

$\text{nonce} \leftarrow\$ \{0,1\}^{192}$

$\tau_\text{🔒} \leftarrow \mathsf{xaenc}(\mathsf{hash}(\mathsf{label}_{\mathsf{cookie}} \| S_r^{\text{pub}}), \text{nonce}, \tau, \text{mac}_1)$

- - - - - - - - - Non-zero $\text{mac}_2$ - - - - - - - -

$\text{mac}_2 \leftarrow \mathsf{mac}(\tau, \text{msg}_\beta)$

Figure 3.1c: Cookie mechanism under load.

- - - - - - - - - First - - - - - - - - - - - - -

$E_i^{\text{pub}}$: $(E_i^{\text{priv}}, E_i^{\text{pub}}) \leftarrow\$ \mathsf{keygen}()$

$C_0 \leftarrow \mathsf{hash}(\text{protocol\_name})$

$H_0 \leftarrow \mathsf{hash}(C_0 \| \text{prologue})$

$H_1 \leftarrow \mathsf{hash}(H_0 \| S_r^{\text{pub}})$

$C_1 \leftarrow \mathsf{hkdf}_1(C_0, E_i^{\text{pub}})$

$H_2 \leftarrow \mathsf{hash}(H_1 \| E_i^{\text{pub}})$

$C_2 \| k_1 \leftarrow \mathsf{hkdf}_2(C_1, \mathsf{dh}(E_i^{\text{priv}}, S_r^{\text{pub}}))$

$S_{i\text{🔒}}^{\text{pub}}$: $S_{i\text{🔒}}^{\text{pub}} \leftarrow \mathsf{aenc}(k_1, 0, S_i^{\text{pub}}, H_2)$

$H_3 \leftarrow \mathsf{hash}(H_2 \| S_{i\text{🔒}}^{\text{pub}})$

$C_3 \| k_2 \leftarrow \mathsf{hkdf}_2(C_2, \mathsf{dh}(S_i^{\text{priv}}, S_r^{\text{pub}}))$

$\text{ts}_\text{🔒}$: $\text{ts}_\text{🔒} \leftarrow \mathsf{aenc}(k_2, 0, \mathsf{timestamp}(), H_3)$

$H_4 \leftarrow \mathsf{hash}(H_3 \| ts_\text{🔒})$

$\text{mac}_1$: $\text{mac}_1 \leftarrow \mathsf{mac}(\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_r^{\text{pub}}), \text{msg}_\alpha)$

- - - - - - - - - Second - - - - - - - - - - - -

$E_r^{\text{pub}}$: $(E_r^{\text{priv}}, E_r^{\text{pub}}) \leftarrow\$ \mathsf{keygen}()$

$C_4 \leftarrow \mathsf{hkdf}_1(C_3, E_r^{\text{pub}})$

$H_5 \leftarrow \mathsf{hash}(H_4 \| E_r^{\text{pub}})$

$C_5 \leftarrow \mathsf{hkdf}_1(C_4, \mathsf{dh}(E_r^{\text{priv}}, E_i^{\text{pub}}))$

$C_6 \leftarrow \mathsf{hkdf}_1(C_5, \mathsf{dh}(E_r^{\text{priv}}, S_i^{\text{pub}}))$

$C_7 \| \pi \| k_3 \leftarrow \mathsf{hkdf}_3(C_6, \text{psk})$

$H_6 \leftarrow \mathsf{hash}(H_5 \| \pi)$

$\text{empty}_\text{🔒}$: $\text{empty}_\text{🔒} \leftarrow \mathsf{aenc}(k_3, 0, \text{empty}, H_6)$

$H_7 \leftarrow \mathsf{hash}(H_6 \| \text{empty}_\text{🔒})$

$\text{mac}_1$: $\text{mac}_1 \leftarrow \mathsf{mac}(\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_i^{\text{pub}}), \text{msg}_\alpha)$

- - - - - - - - - Key Derivation - - - - - - - -

$T^{\rightarrow} \| T^{\leftarrow} \leftarrow \mathsf{hkdf}_2(C_7, \text{empty})$

- - - - - - - - - TransportData - - - - - - - - -

$P_{1\text{🔒}} \leftarrow \mathsf{aenc}(T^{\rightarrow}, N_1^{\rightarrow} = 0, P_1, \text{empty})$

$P_{2\text{🔒}} \leftarrow \mathsf{aenc}(T^{\leftarrow}, N_2^{\leftarrow} = 0, P_2, \text{empty})$

$P_{3\text{🔒}} \leftarrow \mathsf{aenc}(T^{\rightarrow}, N_3^{\rightarrow} = 1, P_3, \text{empty})$

Figure 3.1b: Cryptographic Computations for Protocol Messages.

FIGURE 3.1: (a) An overview of WireGuard's main protocol messages; (b) the cryptographic computations used to create these messages; they need to be adapted accordingly for the receiving side; and (c) the cookie mechanism used by WireGuard to protect hosts against Denial-of-Service attacks. We write $x_\text{🔒}$ for a variable containing an encryption of $x$; $x_\text{🔒}$ is just a variable identifier. The authenticated encryption functions aenc and xaenc take as arguments a key, a nonce, the plaintext to encrypt, and the additional data to authenticate, in this order. $\text{msg}_\alpha$ refers to all the bytes of a message up to but not including $\text{mac}_1$, $\text{msg}_\beta$ is the same but including $\text{mac}_1$. Session key derivation takes places after the second protocol message, symbolised by ↪, at which point the initiator can send messages. The end of the handshake is symbolised by ⇄, after which transport data messages can be sent in both directions. The cookie mechanism is depicted in one direction, initiator to responder, but can actually be used by either initiator or responder, whichever is under load.

MESSAGE EXCHANGE. The protocol begins when $i$ sends the first handshake message to $r$, which includes the following components:

- $I_i$: a fresh session identifier, generated by $i$,
- $E_i^{\text{pub}}$: a fresh ephemeral public key, generated by $i$,
- $S_{i\text{🔒}}^{\text{pub}}$: $i$'s static public key, encrypted for $r$,
- $\text{ts}_{\text{🔒}}$: a timestamp, encrypted with a key that can be computed only by $i$ and $r$, and
- $\text{mac}_1, \text{mac}_2$: message authentication codes (see Section 3.2.2).

In response, $r$ sends the second handshake message containing:

- $I_i$: $i$'s session identifier,
- $I_r$: a fresh session identifier, generated by $r$,
- $E_r^{\text{pub}}$: a fresh ephemeral public key, generated by $r$,
- $\text{empty}_{\text{🔒}}$: an empty bytestring encrypted with a key that can be computed only by $i$ and $r$, and
- $\text{mac}_1, \text{mac}_2$: message authentication codes (see Section 3.2.2).

The encrypted payloads in the two messages serve as authenticators: by computing the corresponding encryption key, each party proves that it knows the private key for its static public key. The encryption key for the second message also requires knowledge of the optional $\text{psk}$ providing an additional authentication guarantee. The two ephemeral keys add fresh session-specific key material that can be used to compute (forward) secret session keys known only to $i$ and $r$.

At the end of these two messages, $i$ and $r$ derive authenticated encryption keys $(T^{\rightarrow}, T^{\leftarrow})$ that can be used to transport IP traffic in the two directions. Importantly, $i$ sends the first transport message, hence confirming the successful completion of the handshake to $r$, before $r$ sends it any encrypted traffic. Each of these transport messages includes:

- $I_i$ or $I_r$: the recipient's session identifier,
- $N_j^{\leftarrow}$ or $N_j^{\rightarrow}$: the current message counter,
- $P_j$: an IP datagram, encrypted under the traffic key.

CRYPTOGRAPHIC COMPUTATIONS. Figure 3.1b describes how each of these message components and traffic keys are computed. As the handshake proceeds, $i$ and $r$ compute a sequence of *transcript hashes* $(H_0, H_1, \ldots, H_7)$ that hashes in all the public data used in the two handshake messages, including:

- protocol_name, prologue: strings identifying the protocol,
- $E_i^{\text{pub}}, E_r^{\text{pub}}$: both ephemeral public keys,
- $S_r^{\text{pub}}, S_{i\text{🔒}}^{\text{pub}}$: both static public keys, but with the initiator's key in encrypted form,
- $ts_{\text{🔒}}, \text{empty}_{\text{🔒}}$: both encrypted handshake payloads, and
- $\pi$: an identifier derived from the pre-shared key.

These transcript hashes serve as unique identifiers for the current stage of the session. In particular, no two completed WireGuard sessions should have the same $H_7$.

Both parties also derive a sequence of *chaining keys* $(C_0, C_1, \ldots, C_7)$ by mixing in all the key material, including:

- protocol_name, $E_i^{\text{pub}}, E_r^{\text{pub}}$,
- $\mathsf{dh}(E_i^{\text{priv}}, S_r^{\text{pub}}) = \mathsf{dh}(S_r^{\text{priv}}, E_i^{\text{pub}})$: the *ephemeral-static* Diffie-Hellman shared secret computed using the initiator's ephemeral key (named first in *ephemeral-static*) and the responder's static key (named second in *ephemeral-static*),
- $\mathsf{dh}(S_i^{\text{priv}}, S_r^{\text{pub}}) = \mathsf{dh}(S_r^{\text{priv}}, S_i^{\text{pub}})$: the static-static shared secret,
- $\mathsf{dh}(E_i^{\text{priv}}, E_r^{\text{pub}}) = \mathsf{dh}(E_r^{\text{priv}}, E_i^{\text{pub}})$: the ephemeral-ephemeral shared secret,
- $\mathsf{dh}(S_i^{\text{priv}}, E_r^{\text{pub}}) = \mathsf{dh}(E_r^{\text{priv}}, S_i^{\text{pub}})$: the static-ephemeral shared secret, and
- $\mathrm{psk}$: the (optional) pre-shared key.

The function dh is the elliptic curve scalar multiplication, taking a private key and a public key as argument, permitting the computation of a *shared secret* [LHT16]. In the preceding list, the initiator uses the first function call, and the responder the second one, respectively.

[LHT16] Langley et al., *Elliptic Curves for Security*

The protocol uses all four combinations of static and ephemeral Diffie-Hellman shared-secret computations to maximally protect against the compromise of some of these keys. The $\mathrm{psk}$ also serves as a defensive countermeasure against quantum adversaries who may be able to break the Diffie-Hellman construction, but not hkdf. Hence, by using a frequently updated $\mathrm{psk}$, WireGuard users can protect current sessions against future quantum adversaries.

Each chaining key is mixed into the next chaining key via an hkdf key derivation that also outputs encryption keys as needed. This chain of key derivations outputs two encryption keys $(k_1, k_2)$ for the first handshake message, an encryption key $(k_3)$ and a PSK identifier $(\pi)$ for the second message, and traffic keys $(T^{\leftarrow}, T^{\rightarrow})$ for all subsequent transport messages.

To encrypt each message, WireGuard uses an authenticated encryption scheme with associated data (AEAD) that takes a key, a counter, a plaintext (padded up to the nearest blocksize) and an optional hash value as associated data. The encryptions in the handshake messages use the current transcript hash $(H_2, H_3, H_6)$ as associated data, which guarantees that the two participants have a consistent session transcript. Transport messages use an empty string as associated data. The message counter is initially set to 0 for each AEAD key and incremented by 1 every time the key is reused.

RELATIONSHIP WITH IKPSK2. The secure channel protocol described above is a direct instantiation of Noise IKpsk2, with five notable differences. First, WireGuard adds local session identifiers $(I_i, I_r)$ for the initiator and responder. Second, WireGuard fixes the payload of the first message to a timestamp, and the one of the second message to the empty string. Third, WireGuard stipulates that the first traffic message is sent from the initiator to the responder. Fourth, WireGuard excludes zero Diffie-Hellman shared secrets to avoid

points of small order, while Noise recommends not to perform this check. Fifth, WireGuard adds two message authentication codes to the handshake messages, to provide stealth and to protect against DoS, as described in the next section. We also observe that although this protocol is superficially similar to other popular Noise protocols like IK (which is used in WhatsApp), there are important differences between these variants and a proof for one does not translate to the other.

### 3.2.2 *Extensions for Stealth and Denial-of-Service*

A VPN protocol operates at a low-level in the networking stack and hence needs to not only protect against cryptographic attacks, but also real-world network-level attacks such as *denial of service* (DoS). Indeed, a cryptographic protocol like IKpsk2 that needs to perform two expensive Diffie-Hellman operations before it can authenticate a handshake message is even more vulnerable to DoS: an adversary can send bogus messages that tie up computing resources on the recipient. A further security goal for WireGuard is that its VPN endpoints should be *stealthy*, in the sense that it should not be possible for a network adversary to blindly scan for WireGuard services.

To support stealthy operation, WireGuard endpoints do not respond to any handshake message unless the sender can prove that it knows the static public key of the recipient. This proof is incorporated in the $\mathrm{mac}_1$ field included in each handshake message, which contains a message authentication code (MAC) computed over the prefix of the current handshake message up to but not including $\mathrm{mac}_1$, using a MAC key derived from the recipient's static public key. The recipient verifies this MAC before processing the message, and stays silent if the MAC fails. Hence, a network adversary who does not know the public key cannot detect whether WireGuard is running on a machine, and at the same time cannot force the recipient to perform two finally useless Diffie-Hellman operations.

To protect more actively against DoS, WireGuard incorporates a cookie-based protocol (depicted in Figure 3.1c) that a host can use when it is under load. For example, if the responder suspects it is under a DoS attack, it can refuse to process the first handshake message and instead send back an initiator-specific fresh *cookie* ($\tau$) that is computed from a frequently rotated secret key ($R_r$) (known only to the responder) and the initiator's IP address ($\mathsf{IP}_i$) and source port ($\mathsf{Port}_i$). The responder encrypts this cookie for the initiator, using a key derived from the initiator's static public key, a fresh nonce, and the $\mathrm{mac}_1$ field of the first message as associated data.

The initiator decrypts $\tau$ and then retries the handshake by sending the first message again, but this time with a second field $\mathrm{mac}_2$ that contains a MAC over the full message up to and including $\mathrm{mac}_1$, using $\tau$ as the MAC key. After verifying this MAC, the responder continues with the standard handshake.

However, to obtain $\tau$, an adversary must be able to read messages on the network path between the initiator and responder and must also know the initiator's static key (which is never sent in the clear by the protocol). And even if the adversary has both these capabilities, it is required to perform session specific cryptographic computations for every handshake message

it sends to the responder, significantly limiting its ability to mount a DoS attack. Hence, this cookie protocol protects the recipient from brute-force network attacks.

Note that the $\mathrm{mac}_2$ field is included in both handshake messages, and hence can be used in both directions, to protect both the initiator and responder from DoS attacks.

The two MACs are WireGuard-specific mechanisms which are not present in IKpsk2. Since they do not use any of the session keys (or hashes or chaining keys) that are used in IKpsk2, adding these mechanisms should, in principle, not affect the security of the secure channel protocol. However, since the static public keys of the two hosts are used in the two MACs, we need to carefully study their impact on the identity-hiding guarantees of IKpsk2.

### 3.2.3 *Instantiating the Cryptographic Algorithms*

WireGuard uses a small set of cryptographic constructions and instantiates them with modern algorithms, carefully chosen to provide strong security as well as high performance:

- dh: all Diffie-Hellman operations use the Curve25519 elliptic curve, which uses 32-byte private and public keys [LHT16];
- hash: all hash operations use the BLAKE2s hash function [SA15], which returns a 32-byte hash;
- aenc: authenticated encryption for handshake and traffic message uses the AEAD scheme ChaCha20Poly1305 [NL18], where the key has 32 bytes, the 96-bit nonce is composed of 32 bits of zeroes followed by the 64-bit little-endian value of the message *counter*, the plaintext is padded with zeroes up to the nearest 16-byte block, and the associated data is a hash value of 32 bytes;
- xaenc: cookie encryption uses an *extended* AEAD construction using XChaCha20Poly1305, which incorporates a 192-bit random nonce into the standard ChaCha20Poly1305 construction [Ber11];
- mac: all MAC operations use the keyed MAC variant of the BLAKE2s hash function, which returns a 16-byte tag;
- $\mathsf{hkdf}_n$: all key derivations use the HKDF construction [KE10], using BLAKE2s as the underlying hash function.

[LHT16] Langley et al., *Elliptic Curves for Security*

[SA15] Saarinen and Aumasson, *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*

[NL18] Nir and Langley, *ChaCha20 and Poly1305 for IETF Protocols*

[Ber11] Bernstein, *Extending the Salsa20 nonce*

[KE10] Krawczyk and Eronen, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*

WireGuard also uses some constants to indicate the specific algorithms it uses and to disambiguate different uses of the mac and xaenc primitives. The protocol_name field is set to the UTF-8 string `"Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s"` while the prologue is set to `"WireGuard v1 zx2c4 Jason@zx2c4.com"`. The $\mathrm{mac}_1$ computation uses the UTF-8 string `"mac1---"` as $\mathsf{label}_{\mathsf{mac1}}$, and the cookie computation uses `"cookie-"` as $\mathsf{label}_{\mathsf{cookie}}$.

### 3.2.4 *Security Goals, Informally*

Using the mechanisms described in this section, WireGuard seeks to provide the following set of strong security guarantees, inheriting the security claims of Noise IKpsk2 [Per18] and extending them with the additional DoS and stealth goals of WireGuard [Don17]. In the following, we use *honest* to refer

[Per18] Perrin, *The Noise Protocol Framework*

[Don17] Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel"

to a party that follows the protocol specification, and *dishonest* to a party that doesn't, i.e. that is controlled by the adversary. Most properties are defined to hold within a *clean* session; we define this notion formally in Section Section 3.5.1.

- **Correctness:** If an honest initiator and an honest responder complete a WireGuard handshake and the messages are not altered by an adversary, then the transport data keys $(T^{\rightarrow}, T^{\leftarrow})$ and the transcript hash $H_7$ are the same on both hosts.
- **Secrecy:** If a transport data message $P$ is sent over a tunnel between two honest hosts, then this message is kept confidential from the adversary. Furthermore, the traffic keys for this tunnel are also confidential.
- **Forward Secrecy:** Secrecy for a session holds even if both the static private keys $(S_i^{\mathrm{priv}}, S_r^{\mathrm{priv}})$ and the pre-shared key $(\mathrm{psk})$ become known to the adversary, but only after the session has been completed and all its traffic keys and chaining keys are deleted by both parties.

  Secrecy also holds even if the static and ephemeral keys are compromised (e.g. by a quantum adversary), as long as the pre-shared key is not compromised.
- **Mutual Authentication:** If an honest initiator (resp. responder) completed a handshake (ostensibly) with an honest peer, then that peer must have participated in this handshake. Moreover, if a host $A$ receives a plaintext message over a WireGuard tunnel that claims to be from host $B$, then $B$ must have (intentionally) sent this message to $A$.
- **Resistance against Key Compromise Impersonation (KCI):** The recipient of a message can authenticate the message's sender even if the recipient's static key is compromised.
- **Resistance against Identity Mis-Binding:** If two honest parties derive the same traffic keys in some WireGuard session, then they agree on each other's identities, even if one or both of them have been interacting with a dishonest party or a honest party with compromised keys. This property is also called resistance against unknown key-share attacks.
- **Resistance against Replay:** Any protocol message sent may be accepted at most once by the recipient.
- **Session Uniqueness:** There is at most one honest initiator session and at most one honest responder session for a given traffic key. Similarly, there is at most one honest initiator session and at most one honest responder session for given handshake messages.
- **Channel Binding:** Two sessions that have the same final session transcript hash $H_7$ share the same view and the same session keys.
- **Identity Hiding:** Just by looking at the messages transmitted over the network, a passive adversary cannot infer the static keys involved in a session. (However, these identities are not forward secret: If the responder's static key gets compromised, the adversary can later decrypt the initiator's static public key that was transmitted in the first message.)

- **DoS Resistance:** The adversary cannot have a message accepted by a recipient under load without having first made a round trip with that recipient. In practice, this means that the adversary has to be at the claimed address. Because we assume that the adversary controls the network, we cannot prove more than enforcing a round trip.

The security goals above are stated in terms of completed WireGuard sessions, with most security guarantees only applying after the third message, when both initiator and responder start freely sending and receiving data. In particular, the first transport data message (i.e. the third message) serves as key confirmation to the responder, and is needed to prove that the initiator has control over its ephemeral key. This is why, in WireGuard, the responder does not send any data until it sees this third message. In the rest of this chapter, we investigate whether WireGuard achieves the goals set out above.

## 3.3 Cryptographic Assumptions

This section presents the assumptions that we make on the cryptographic primitives used by WireGuard. For most primitives, the desired assumption is already present in the library of primitives of CryptoVerif, so we just need to call a macro to use that assumption. Still, we had to design a new model for Curve25519, detailed below.

### 3.3.1 *Random Oracle Model*

We assume that BLAKE2s is a random oracle [BR93]. This assumption is justified in [LMN16] using a weak ideal block cipher. In particular, BLAKE2s uses a prefix-free Merkle-Damgård construction, thanks to the use of finalisation flags. Therefore, extension attacks which apply to pure Merkle-Damgård constructions do not apply to BLAKE2s.

[BR93] Bellare and Rogaway, "Random Oracles Are Practical: a Paradigm for Designing Efficient Protocols"

[LMN16] Luykx et al., "Security Analysis of BLAKE2s Modes of Operation"

### 3.3.2 *IND-CPA and INT-CTXT for AEAD*

We assume that the ChaCha20Poly1305 AEAD scheme [NL18] is IND-CPA (indistinguishable under chosen plaintext attacks) and INT-CTXT (ciphertext integrity) [BN00], provided the same nonce is never used twice with the same key. IND-CPA means that the adversary has a negligible probability of distinguishing encryptions of two distinct messages of the same length that it has chosen. INT-CTXT means that an adversary with access to encryption and decryption oracles has a negligible probability of forging a ciphertext that decrypts successfully and has not been returned by the encryption oracle. These properties are justified in [Pro14], assuming ChaCha20 is a PRF (pseudo-random function) and Poly1305 is an $\epsilon$-almost-$\Delta$-universal hash function. The latter property is shown to hold in [Ber05].

[NL18] Nir and Langley, *ChaCha20 and Poly1305 for IETF Protocols*

[BN00] Bellare and Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm"

[Pro14] Procter, *A Security Analysis of the Composition of ChaCha20 and Poly1305*

[Ber05] Bernstein, "The Poly1305-AES Message-Authentication Code"

### 3.3.3 *Curve25519 and Gap Diffie-Hellman*

WireGuard uses the elliptic curve Curve25519 [LHT16] for Diffie-Hellman key exchanges. Curve25519, as implemented in WireGuard and as specified by RFC 7748 [LHT16], satisfies the following properties:

[LHT16] Langley et al., *Elliptic Curves for Security*

1. It is an elliptic curve defined by an equation of the form $Y^2 = X^3 + AX^2 + X$ in the field $\mathbb{F}_p$ of non-zero integers modulo a large prime $p$, where $A^2 - 4$ is not a square modulo $p$.
2. It forms a commutative group of order $kq$ where $k$ (cofactor) is a small integer and $q$ is a large prime, using point addition as group law and the point at infinity $\infty$ as neutral element. The base point $g_0$ is a point on the curve with prime order $q$.
3. The incoming public keys are not verified and the implementation uses a single coordinate ladder, that is, the curve points are only represented by their $X$ coordinate. When $X^3 + AX^2 + X$ is a square $Y^2$, $X$ represents the curve point $(X, Y)$ or $(X, -Y)$. When $X^3 + AX^2 + X$ is not a square, $X$ does not represent a point on the curve, but on its quadratic twist. The twist is also an elliptic curve, which forms a group of order $k'q'$ where $k'$ is a small integer and $q'$ is a large prime, using point addition as group law and the point at infinity $\infty$ as neutral element. (In particular, the incoming public keys can represent any point on the curve or its twist, and may not belong to the subgroup generated by the base point $g_0$.)
4. The cofactor $k$ of the curve is a multiple of the cofactor $k'$ of the twist.
5. The single coordinate ladder is defined following [Ber06, Theorem 2.1]: We consider the elliptic curve $E(\mathbb{F}_{p^2})$ defined by the equation $Y^2 = X^3 + AX^2 + X$ in a quadratic extension $\mathbb{F}_{p^2}$ of $\mathbb{F}_p$, we define $X_0 : E(\mathbb{F}_{p^2}) \to \mathbb{F}_{p^2}$ by $X_0(\infty) = 0$ and $X_0(X, Y) = X$, and for $X \in \mathbb{F}_p$ and $y$ an integer, we define $y \cdot X \in \mathbb{F}_p$ as $y \cdot X = X_0(yQ)$ for all $Q \in E(\mathbb{F}_{p^2})$ such that $X_0(Q) = X$.
6. The public keys (bitstrings in a finite set $G$) are mapped to elements of $\mathbb{F}_p$ by the function $\mathsf{decode\_pk} : G \to \mathbb{F}_p$ and conversely, elements of $\mathbb{F}_p$ are mapped to public keys by the function $\mathsf{encode\_pk} : \mathbb{F}_p \to G$, such that $\mathsf{decode\_pk} \circ \mathsf{encode\_pk}$ is the identity.
7. The Diffie-Hellman "exponentiation" $\exp : G \times \mathbb{Z} \to G$ is defined by
$$\exp(X, y) = \mathsf{encode\_pk}(y \cdot \mathsf{decode\_pk}(X))$$
8. The private keys are chosen uniformly in $\{kn \mid n \in S\}$ where $S \subseteq \{n_{\min}, \ldots, n_{\max}\}$, $n_{\min} < n_{\max}$, $n_{\max} - n_{\min} < q$, $n_{\max} - n_{\min} < q'$, and all elements of $S$ are prime to $qq'$.

[Ber06] Bernstein, "Curve25519: New Diffie-Hellman Speed Records"

Curve25519 satisfies these properties with $p = 2^{255} - 19$, $k = 8$, $k' = 4$, $q = 2^{252} + \delta$ with $0 < \delta < 2^{128}$, $q' = 2^{253} - 9 - 2\delta$, $S = \{2^{251}, \ldots, 2^{252} - 1\}$.[3] The set $G$ of public keys consists of bitstrings of 32 bytes, or equivalently $G = \{0, \ldots, 2^{256} - 1\}$. The function $\mathsf{decode\_pk} : G \to \mathbb{F}_p$ is defined by $\mathsf{decode\_pk}(X) = (X \bmod 2^{255}) \bmod p$. Conversely, $\mathsf{encode\_pk} : \mathbb{F}_p \to G$ is such that $\mathsf{encode\_pk}(X)$ is the representation of $X$ as an element of $\{0, \ldots, p-1\}$. For generality, our model supports not only Curve25519, but any elliptic curve that satisfies assumptions 1 to 8 above.

[3] The exact value of $\delta$ is `0x14def9dea2f79cd65812631a5cf5d3ed`

Let us first establish a few properties of $y \cdot X$.

**Lemma 3.1.** *1. We can define $y \cdot X$ for $y$ in $\mathbb{Z}_{kqq'}$.*

*2. $y \cdot (z \cdot X) = (yz) \cdot X$.*

3. *Let $Z$ be the set of integers multiple of $k$ and prime to $qq'$ modulo $kqq'$. For any $z \in Z$, for any $X, Y \in \mathbb{F}_p$, we have $z \cdot X = z \cdot Y$ if and only if $k \cdot X = k \cdot Y$.*
4. *Let $G_{\mathrm{sub}} = \{k \cdot X \mid X \in \mathbb{F}_p\}$. For any $z$ prime to $qq'$, for any $X, Y \in G_{\mathrm{sub}}$, we have $z \cdot X = z \cdot Y$ if and only if $X = Y$.*

*Proof.* *1.* We have $kqq'Q = \infty$ for any $Q$: if $Q$ is on the curve, then $kqQ = \infty$ since $\infty$ is the neutral element of the curve and its order is $kq$, so $kqq'Q = \infty$; if $Q$ is on the twist, then $k'q'Q = \infty$ since $\infty$ is the neutral element of the twist and its order is $k'q'$, so $kqq'Q = \infty$ since $kqq'$ is a multiple of $k'q'$. Hence, $(y + nkqq')Q = yQ$ for any $n$, so we can define $y \cdot X$ when $y$ is in $\mathbb{Z}_{kqq'}$.

*2.* We have $y \cdot (z \cdot X) = X_0(yQ')$ for all $Q' \in E(\mathbb{F}_{p^2})$ such that $X_0(Q') = z \cdot X = X_0(zQ)$ for all $Q \in E(\mathbb{F}_{p^2})$ such that $X_0(Q) = X$. Taking $Q' = zQ$, we have $y \cdot (z \cdot X) = X_0(yzQ)$ for all $Q \in E(\mathbb{F}_{p^2})$ such that $X_0(Q) = X$, so $y \cdot (z \cdot X) = (yz) \cdot X$.

*3.* We have $z = kz'$ for some $z'$ prime to $qq'$: there exist $z''$ and $n$ such that $z'z'' + nqq' = 1$. Then $kz'z'' + nkqq' = k$. Hence, $z'' \cdot (z \cdot X) = (zz'') \cdot X = (kz'z'') \cdot X = k \cdot X$ and similarly $z'' \cdot (z \cdot Y) = k \cdot Y$ .

If $k \cdot X = k \cdot Y$, then $z \cdot X = (kz') \cdot X = z' \cdot (k \cdot X) = z' \cdot (k \cdot Y) = (kz') \cdot Y = z \cdot Y$. Conversely, suppose $z \cdot X = z \cdot Y$. Thus, $z'' \cdot (z \cdot X) = z'' \cdot (z \cdot Y)$, so $k \cdot X = k \cdot Y$.

*4.* We have $X = k \cdot X'$ and $Y = k \cdot Y'$ for some $X'$ and $Y'$. By *3.* applied to $kz$, $X'$, $Y'$, we have $(kz) \cdot X' = (kz) \cdot Y'$ if and only if $k \cdot X' = k \cdot Y'$. That is exactly $z \cdot X = z \cdot Y$ if and only if $X = Y$. □

Property *2* implies that $\exp(\exp(g, y), z) = \mathsf{encode\_pk}(z \cdot (y \cdot X_0(g_0))) = \mathsf{encode\_pk}((zy) \cdot X_0(g_0))$, where $g = \mathsf{encode\_pk}(X_0(g_0))$ represents the base point. Hence, by commutativity of integer multiplication, $\exp(\exp(g, y), z) = \exp(\exp(g, z), y)$: the same Diffie-Hellman shared secret is computed by both participants of the protocol.

We say that public keys $X$ and $Y$ such that $k \cdot \mathsf{decode\_pk}(X) = k \cdot \mathsf{decode\_pk}(Y)$ are *equivalent*, because they yield the same Diffie-Hellman shared secrets as shown by Lemma 3.1, property *3*. There are in general several public keys equivalent to a public key $X$. Moreover, the public keys may be 0, and $x \cdot 0 = 0$ for all $x$.

While most proofs of Diffie-Hellman key agreements assume a prime order group, that assumption is not correct for most implementations of Curve25519. For instance, the identity mis-binding issue that we discuss in Section 3.6 would not appear in a prime order group. Therefore, we need to provide a new model that takes into account the properties mentioned above.

In CryptoVerif, we first define the following types:

```
type G [bounded, large].
type G_sub [bounded, large].
type Z [bounded, large, nonuniform].
```

The type $G$ represents the set of public keys; it is bounded because it is represented by bitstrings of bounded length, and large because collisions between randomly chosen elements in $G$ have a small probability. (The

set $G$ contains at least $p$ elements since encode_pk is injective, and $p$ is a large prime.) The type $G_{\mathrm{sub}}$ represents the set $\{k \cdot X \mid X \in \mathbb{F}_p\} = \{X_0(Q) \mid Q$ is in the subgroup of order $q$ of the curve or in the subgroup of order $q'$ of the twist$\}$. The type $Z$ corresponds to the set $Z$ defined in Lemma 3.1. When honest participants choose private keys, they are chosen uniformly in a *subset* of $Z$, $\{kn \mid n \in S\}$, considered modulo $kqq'$. By hypothesis, $n \in S$ is prime to $qq'$. Moreover, since $k$ is small integer and $q$ and $q'$ are large primes, $k$ is not a multiple of $q$ nor $q'$, so $k$ is also prime to $qq'$. Hence, $kn$ is a multiple of $k$ and prime to $qq'$, so $kn$ considered modulo $kqq'$ is in $Z$. Since these elements do not cover the whole set $Z$, the distribution for choosing random private keys inside the whole $Z$ is non-uniform, which is indicated by the annotation nonuniform.

The main idea of our model is to rely on a Diffie-Hellman assumption in $G_{\mathrm{sub}}$, and so to work as much as possible with elements in $G_{\mathrm{sub}}$. We rewrite the computations in $G$ into computations in $G_{\mathrm{sub}}$ by first mapping the public keys $X \in G$ to $k \cdot \mathsf{decode\_pk}(X) \in G_{\mathrm{sub}}$.

We define functions:

$\mathrm{fun}\ \mathrm{exp}(G, Z) : G.$
$\mathrm{fun}\ \mathrm{mult}(Z, Z) : Z.$
$\mathrm{equation}\ \mathrm{builtin}\ \mathsf{commut}(\mathrm{mult}).$

We let $\mathrm{exp}(X, y) = \mathsf{encode\_pk}(y \cdot \mathsf{decode\_pk}(X))$ and $\mathrm{mult}$ be the product modulo $kqq'$, in $Z$. Since its two arguments are multiples of $k$ and prime to $qq'$, so is its result, and it is in $Z$. The last line states that the function $\mathrm{mult}$ is commutative. (We could add associativity and other algebraic properties, but commutativity is typically sufficient to prove security of basic Diffie-Hellman key exchanges. More algebraic properties may be needed to prove group Diffie-Hellman protocols, for instance. Note that not modelling these does not restrict the adversary in the computational model.)

$\mathrm{fun}\ \mathrm{pow\_k}(G) : G_{\mathrm{sub}}.$
$\mathrm{fun}\ \mathrm{exp\_div\_k}(G_{\mathrm{sub}}, Z) : G_{\mathrm{sub}}.$
$\mathrm{fun}\ \mathrm{G_{sub}2G}(G_{\mathrm{sub}}) : G\ [\mathsf{data}].$
$\mathrm{equation}\ \mathrm{forall}\ X : G_{\mathrm{sub}}, X' : G_{\mathrm{sub}};$
$\quad (\mathrm{pow\_k}(\mathrm{G_{sub}2G}(X)) = \mathrm{pow\_k}(\mathrm{G_{sub}2G}(X'))) = (X = X').$

We have $\mathrm{pow\_k}(X) = k \cdot \mathsf{decode\_pk}(X)$, and it is in $G_{\mathrm{sub}}$ for all $X$ in $G$. We have $\mathrm{exp\_div\_k}(X, y) = (y/k) \cdot X$. This function is convenient since the private keys in $Z$ are always multiples of $k$. Let us show that it is well defined and operates on $G_{\mathrm{sub}}$. Since $y$ is a multiple of $k$, there exists $y'$ such that $y = ky'$. There are $k$ representatives of $y/k$ modulo $kqq'$, $y' + nqq'$ for $n \in \{0, \ldots, k-1\}$, but all representatives yield the same value for $(y/k) \cdot X$: since $X \in G_{\mathrm{sub}}$, there exists $X'$ such that $X = k \cdot X'$, so $(y' + nqq') \cdot X = (y' + nqq') \cdot (k \cdot X') = (ky' + nkqq') \cdot X' = y \cdot X'$. Moreover, this value is equal to $k \cdot (y' \cdot X')$, so it is in $G_{\mathrm{sub}}$. The function $\mathrm{G_{sub}2G}$ is encode_pk restricted to $G_{\mathrm{sub}}$; it converts elements of type $G_{\mathrm{sub}}$ to type $G$. The annotation data tells CryptoVerif that it is injective. The last equation says that $\mathrm{pow\_k} \circ \mathrm{G_{sub}2G}$ is injective. The function $\mathsf{decode\_pk} \circ \mathrm{G_{sub}2G}$ is the identity, so $\mathrm{pow\_k}(\mathrm{G_{sub}2G}(X)) = k \cdot X$ and similarly $\mathrm{pow\_k}(\mathrm{G_{sub}2G}(X')) = k \cdot X'$. By Lemma 3.1, property *4*, $k \cdot X = k \cdot X'$ if and only if $X = X'$.

We also define constants:

$$\begin{aligned}
&\text{const zero} : G.\\
&\text{const zero}_{\text{sub}} : G_{\text{sub}}.\\
&\text{equation zero} = \text{G}_{\text{sub}}\text{2G}(\text{zero}_{\text{sub}}).\\
&\text{const } g : G.\\
&\text{const g_k} : G_{\text{sub}}.\\
&\text{equation pow_k}(g) = \text{g_k}.\\
&\text{equation g_k} \neq \text{zero}_{\text{sub}}.
\end{aligned}$$

The constant 0 is zero as an element of $G$ and $\text{zero}_{\text{sub}}$ as an element of $G_{\text{sub}}$. The constant $g = \mathsf{encode\_pk}(X_0(g_0))$ represents the base point, and $\text{g\_k} = k \cdot \text{decode\_pk}(g)$.

We also state equations that hold on these functions:

$$\begin{aligned}
&\text{equation forall } X : G, y : Z;\\
&\quad \exp(X, y) \qquad (3.1)\\
&= \text{G}_{\text{sub}}\text{2G}(\text{exp_div_k}(\text{pow_k}(X), y)).
\end{aligned}$$

$$\begin{aligned}
&\text{equation forall } X : G_{\text{sub}}, y : Z, z : Z;\\
&\quad \text{exp_div_k}(\text{pow_k}(\text{G}_{\text{sub}}\text{2G}(\text{exp_div_k}(X, y))), z) \qquad (3.2)\\
&= \text{exp_div_k}(X, \text{mult}(y, z)).
\end{aligned}$$

Equation (3.1) follows from $y \cdot \text{decode\_pk}(X) = (y/k) \cdot (k \cdot \text{decode\_pk}(X))$ and Equation (3.2) from $(z/k) \cdot k \cdot (y/k) \cdot X = (yz/k) \cdot X$. Equation (3.2) applies in particular to simplify $\exp(\exp(X, y), z)$ after applying (3.1):

$$\begin{aligned}
&\exp(\exp(X, y), z)\\
&= \text{G}_{\text{sub}}\text{2G}(\text{exp_div_k}(\text{pow_k}(\text{G}_{\text{sub}}\text{2G}(\text{exp_div_k}(\text{pow_k}(X), y))), z))\\
&= \text{G}_{\text{sub}}\text{2G}(\text{exp_div_k}(\text{pow_k}(X), \text{mult}(y, z)))
\end{aligned}$$

This equation with $X = g$, combined with the commutativity of mult, shows that both participants of the protocol compute the same Diffie-Hellman shared secret. These equations are used by CryptoVerif as rewrite rules, to rewrite the left-hand side into the right-hand side. They allow to rewrite computations in $G$ into computations that happen in $G_{\text{sub}}$, after mapping the public keys $X \in G$ to $k \cdot \text{decode\_pk}(X) \in G_{\text{sub}}$. In particular, $\exp(g, y) = \text{G}_{\text{sub}}\text{2G}(\text{exp\_div\_k}(\text{g\_k}, y))$ and $\exp(\exp(g, y), z) = \text{G}_{\text{sub}}\text{2G}(\text{exp\_div\_k}(\text{g\_k}, \text{mult}(y, z)))$.

The next equations allow CryptoVerif to simplify equality tests with 0, which are used by some protocols, including WireGuard, to exclude elements of low order from the allowed public keys.

$$\begin{aligned}
&\text{equation forall } X : G_{\text{sub}}, y : Z; (\text{exp_div_k}(X, y) = \text{zero}_{\text{sub}}) = (X = \text{zero}_{\text{sub}}).\\
&\text{equation forall } X : G_{\text{sub}}, y : Z; (\text{exp_div_k}(X, y) \neq \text{zero}_{\text{sub}}) = (X \neq \text{zero}_{\text{sub}}).\\
&\text{equation forall } X : G_{\text{sub}}; (\text{pow_k}(\text{G}_{\text{sub}}\text{2G}(X)) = \text{zero}_{\text{sub}}) = (X = \text{zero}_{\text{sub}}).\\
&\text{equation forall } X : G_{\text{sub}}; (\text{pow_k}(\text{G}_{\text{sub}}\text{2G}(X)) \neq \text{zero}_{\text{sub}}) = (X \neq \text{zero}_{\text{sub}}).
\end{aligned}$$

When $y \in Z$, $y = ky'$ for some $y'$ prime to $qq'$. Moreover, $y' \cdot 0 = 0$. Therefore, $(y/k) \cdot X = 0$ if and only if $y' \cdot X = y' \cdot 0$ if and only if $X = 0$ by Lemma 3.1, property *4*. We have $\text{pow\_k}(\text{G}_{\text{sub}}\text{2G}(X)) = k \cdot X$ and $k$ is

prime to $qq'$, so $k \cdot X = 0$ if and only if $k \cdot X = k \cdot 0$ if and only if $X = 0$ by Lemma 3.1, property *4*.

Other properties serve to simplify equalities between Diffie-Hellman values in $G_{\mathrm{sub}}$, with the goal of showing that these equalities are false. When the Diffie-Hellman shared secrets are passed to a random oracle, these equality tests appear after using the random oracle assumption: we compare the arguments of each call to the random oracle with arguments of previous calls, to know whether the random oracle should return the result of a previous call.

$$\text{equation forall } X : G_{\mathrm{sub}}, X' : G_{\mathrm{sub}}, y : Z; \\ (\mathrm{exp\_div\_k}(X, y) = \mathrm{exp\_div\_k}(X', y)) = (X = X'). \tag{3.3}$$

$$\text{equation forall } X : G_{\mathrm{sub}}, x' : Z, y : Z; \\ (\mathrm{exp\_div\_k}(X, y) = \mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x', y))) = \\ (X = \mathrm{pow\_k}(\mathrm{G_{sub}2G}(\mathrm{exp\_div\_k}(\mathrm{g\_k}, x')))). \tag{3.4}$$

$$\text{collision } y \xleftarrow{R} Z; z \xleftarrow{R} Z; [\text{random\_choices\_may\_be\_equal}] \text{ forall } X : G_{\mathrm{sub}}; \\ \mathrm{return}(\mathrm{exp\_div\_k}(X, y) = \mathrm{exp\_div\_k}(X, z)) \\ \approx_{\mathrm{Pcoll1rand}(Z)} \mathrm{return}((X = \mathrm{zero_{sub}}) \vee (y = z)). \tag{3.5}$$

$$\text{collision } x \xleftarrow{R} Z; \text{forall } X : G_{\mathrm{sub}}, Y : G_{\mathrm{sub}}; \\ \mathrm{return}(\mathrm{exp\_div\_k}(X, x) = Y) \\ \approx_{2\times\mathrm{Pcoll1rand}(Z)} \\ \mathrm{return}((X = \mathrm{zero_{sub}}) \wedge (Y = \mathrm{zero_{sub}})) \\ \text{if } X \text{ independent-of } x \wedge Y \text{ independent-of } x. \tag{3.6}$$

$$\text{collision } x \xleftarrow{R} Z; \text{forall } y : Z, X : G_{\mathrm{sub}}; \\ \mathrm{return}(\mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, y)) = X) \approx_{2\times\mathrm{Pcoll1rand}(Z)} \mathrm{return}(\mathsf{false}) \\ \text{if } y \text{ independent-of } x \wedge X \text{ independent-of } x. \tag{3.7}$$

$$\text{collision } x \xleftarrow{R} Z; y \xleftarrow{R} Z; [\text{random\_choices\_may\_be\_equal}] \text{ forall } X : G_{\mathrm{sub}}; \\ \mathrm{return}(\mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, y)) = X) \approx_{4\times\mathrm{Pcoll1rand}(Z)} \mathrm{return}(\mathsf{false}) \\ \text{if } X \text{ independent-of } x \vee X \text{ independent-of } y. \tag{3.8}$$

$$\text{collision } x \xleftarrow{R} Z; y \xleftarrow{R} Z; y' \xleftarrow{R} Z; [\text{random\_choices\_may\_be\_equal}] \\ \mathrm{return}(\mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, y)) = \mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, y'))) \\ \approx_{\mathrm{Pcoll1rand}(Z)} \mathrm{return}(y = y'). \tag{3.9}$$

Equation (3.3) follows from Lemma 3.1, property *4* because $y = ky'$ for some $y'$ prime to $qq'$. In particular, using (3.1), injectivity of $\mathrm{G_{sub}2G}$, and (3.3), $\exp(X, y) = \exp(X', y)$ simplifies into $\mathrm{pow\_k}(X) = \mathrm{pow\_k}(X')$. In contrast, in a prime order group, $\exp(X, y) = \exp(X', y)$ implies $X = X'$. This is the reason why, in the identity mis-binding issue of Section 3.6, we fail to prove equality of the public keys $X = X'$ and can only prove $\mathrm{pow\_k}(X) = \mathrm{pow\_k}(X')$.

Equation (3.4) is a particular case of (3.3) when $X' = x' \cdot \mathrm{g\_k} = \mathrm{pow\_k}(\mathrm{G_{sub}2G}(\mathrm{exp\_div\_k}(\mathrm{g\_k}, x')))$. However, CryptoVerif would not apply (3.3) to a term of the form $\mathrm{exp\_div\_k}(X, y) = \mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x', y))$.

Let us solve the equation $y \cdot X = z \cdot X$ for $X \in G_{\mathrm{sub}}$. If $X = 0$, then $y \cdot X = 0 = z \cdot X$ for all $y$ and $z$. Otherwise, $X = X_0(Q)$ where $Q$ is either in the subgroup of order $q$ of the curve or in the subgroup of order $q'$ of the twist. Since $X \neq 0$, $Q \neq \infty$, so $Q$ is a generator of either of these subgroups. The equation $y \cdot X = z \cdot X$ means $X_0(yQ) = X_0(zQ)$, that is, $yQ = zQ$ or $yQ = -zQ$. If $Q$ is on the curve, this means $y \equiv z \mod q$ or $y \equiv -z \mod q$. If $Q$ is on the twist, this means $y \equiv z \mod q'$ or $y \equiv -z \mod q'$. In other words, $z + mq$ and $-z + mq$ are *equivalent private keys* for all $m$ and all public keys in $G_{\mathrm{sub}}$ that correspond to points on the curve, and $z + mq'$ and $-z + mq'$ are equivalent private keys for all $m$ and all public keys in $G_{\mathrm{sub}}$ that correspond to points on the twist. (In the case of Curve25519, there are indeed honestly generated equivalent private keys: $kn$ and $k(q-n)$ for $n \in \{2^{251}, \ldots, 2^{251} + \delta\}$ can both be honestly generated since $q - n \in \{2^{251}, \ldots, 2^{251} + \delta\}$ in this case. Similarly, $kn$ and $k(q'-n)$ for $n \in \{2^{252} - 8 - 2\delta, \ldots, 2^{252} - 1\}$ can both be honestly generated since $q' - n \in \{2^{252} - 8 - 2\delta, \ldots, 2^{252} - 1\}$.)

In the collision statement (3.5), Pcoll1rand($Z$) is the probability that a randomly chosen element $x$ in $Z$ is equal to an element of $Z$ independent of $x$. Since random private keys are chosen uniformly among a set of $|S|$ elements, Pcoll1rand($Z$) $= 1/|S|$. For Curve25519, Pcoll1rand($Z$) $= 2^{-251}$. Statement (3.5) means that the probability of distinguishing $\mathrm{exp\_div\_k}(X, y) = \mathrm{exp\_div\_k}(X, z)$ from $(X = \mathrm{zero}_{\mathrm{sub}}) \vee (y = z)$ is at most Pcoll1rand($Z$), assuming $y$ and $z$ are chosen randomly in $Z$ ($y \xleftarrow{R} Z; z \xleftarrow{R} Z$). The annotation [random_choices_may_be_equal] means that the random choices $y \xleftarrow{R} Z$ and $z \xleftarrow{R} Z$ may be either independent or the same random choice. (Without this annotation, they would necessarily be independent.) Grouping the two cases in a single collision statement allows CryptoVerif to apply it even when it cannot determine whether the random choices are independent or identical: when $x$ and $y$ are two cells of the same array with indices that may be different or equal. In (3.5), when $X = 0$ or $y = z$, both sides are true. Suppose now that $X \neq 0$ and $y \neq z$. Then, $y$ and $z$ are independent random choices, and the right-hand side is false. The expressions differ when the left-hand side is true, that is, $(y/k) \cdot X = (z/k) \cdot X$, so $y/k \equiv z/k \mod q$ or $y/k \equiv -z/k \mod q$ when $X = X_0(Q)$ for $Q$ on the curve and $y/k \equiv z/k \mod q'$ or $y/k \equiv -z/k \mod q'$ when $X = X_0(Q)$ for $Q$ on the twist. Since $q$ and $q'$ are greater than $n_{\max} - n_{\min}$ and $y/k$ and $z/k$ are in $S \subseteq \{n_{\min}, \ldots, n_{\max}\}$, the only possibility for $y/k \equiv z/k \mod q$ or $y/k \equiv z/k \mod q'$ is $y = z$, which is excluded, and the equations $y/k \equiv -z/k \mod q$ and $y/k \equiv -z/k \mod q'$ each have at most one solution for $z$ once $y$ is fixed, so they hold with probability at most Pcoll1rand($Z$). Therefore, the two expressions differ with at most probability Pcoll1rand($Z$).

Statement (3.6) means that the probability of distinguishing $\mathrm{exp\_div\_k}(X, x) = Y$ from $(X = \mathrm{zero}_{\mathrm{sub}}) \wedge (Y = \mathrm{zero}_{\mathrm{sub}})$ is at most $2 \times$ Pcoll1rand($Z$) assuming $x$ is chosen randomly in $Z$ ($x \xleftarrow{R} Z$) and $X$ and $Y$ are independent of $x$. Indeed, suppose that $\mathrm{exp\_div\_k}(X, x) = Y$ differs from $(X = \mathrm{zero}_{\mathrm{sub}}) \wedge (Y = \mathrm{zero}_{\mathrm{sub}})$. If $X = 0$, then $(x/k) \cdot X = 0$, so both expressions reduce to $Y = 0$, so they cannot differ. Therefore, $X \neq 0$. The second expression is then false. If $(x/k) \cdot X = Y$, then $Y = y \cdot X$ for some

$y$ independent of $x$. Moreover $x = kn$ for $n \in S$ chosen randomly, and $y$ is independent of $n$. The equality $(x/k) \cdot X = n \cdot X = Y = y \cdot X$ holds if and only if $n \equiv y \mod q$ or $n \equiv -y \mod q$ when $X = X_0(Q)$ for $Q$ on the curve and $n \equiv y \mod q'$ or $n \equiv -y \mod q'$ when $X = X_0(Q)$ for $Q$ on the twist. Each of these equations has at most one solution for $n$ since $q$ and $q'$ are greater than $n_{\max} - n_{\min}$, so there are at most two solutions for $n$ in total (for Curve25519, there are indeed two solutions for some values of $y$, due to the existence of equivalent private keys), hence the first expression $(x/k) \cdot X = Y$ is true with probability at most $2 \times \mathrm{Pcoll1rand}(Z)$, and the two expressions differ with at most that probability. The support for side-conditions in collision statements is an extension of CryptoVerif that we implemented.

In statement (3.7), when the equality $\mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, y)) = X$ holds, we have $X = \mathrm{exp\_div\_k}(\mathrm{g\_k}, z)$ for some $z \in Z$ independent of $x$. Hence, since $\mathrm{g\_k}$ corresponds to a point on the curve, $xy/k \equiv z/k \mod q$ or $xy/k \equiv -z/k \mod q$. Since $y/k$ is prime to $qq'$, so prime to $q$, $y/k$ is invertible modulo $q$, so $x \equiv z/y \mod q$ or $x \equiv -z/y \mod q$, and $z/y$ is independent of $x$, so these equalities happen with probability at most $2 \times \mathrm{Pcoll1rand}(Z)$.

When $x$ and $y$ are independent, statement (3.8) is a consequence of statement (3.7). Suppose that $x$ and $y$ are the same random choice: $x = y$. When the equality $\mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, x)) = X$ holds, we have $X = \mathrm{exp\_div\_k}(\mathrm{g\_k}, z)$ for some $z \in Z$ independent of $x$. Hence, since $\mathrm{g\_k}$ corresponds to a point on the curve, $x^2/k \equiv z/k \mod q$ or $x^2/k \equiv -z/k \mod q$, that is, $x^2 \equiv z \mod q$ or $x^2 \equiv -z \mod q$, and $z$ is independent of $x$. Each of these equations has at most 2 solutions for $x$, so these equalities happen with probability at most $4 \times \mathrm{Pcoll1rand}(Z)$. (Furthermore, for Curve25519, $q \equiv 1 \mod 4$, so $-1$ is a square modulo $q$, hence both $z$ and $-z$ are squares simultaneously, so $x^2 \equiv z \mod q$ or $x^2 \equiv -z \mod q$ may have 4 solutions for $x$ in total.)

Statement (3.9) holds when $y$ and $y'$ are the same random choice. When $y$ and $y'$ are independent random choices and $\mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, y)) = \mathrm{exp\_div\_k}(\mathrm{g\_k}, \mathrm{mult}(x, y'))$, since $\mathrm{g\_k}$ corresponds to a point on the curve, we have $xy/k \equiv xy'/k \mod q$ or $xy/k \equiv -xy'/k \mod q$. Since $x$ is prime to $qq'$, so prime to $q$, $x$ is invertible modulo $q$, so $y/k \equiv y'/k \mod q$ or $y/k \equiv -y'/k \mod q$. In the first case, since $q$ is greater than $n_{\max} - n_{\min}$ and $y/k$ and $y'/k$ are in $S \subseteq \{n_{\min}, \ldots, n_{\max}\}$, we have $y = y'$. Hence the two sides of (3.9) differ only in the second case, which happens with probability less than $\mathrm{Pcoll1rand}(Z)$.

For simplicity, we omit 4 additional collision statements, which can be inferred from the ones above. CryptoVerif may be able to infer some of them automatically in the future. When CryptoVerif transforms a game using a Diffie-Hellman assumption, it renames $\mathrm{exp\_div\_k}$ into $\mathrm{exp\_div\_k}'$, in order to prevent the repeated application of the same game transformation. Hence, we define a symbol $\mathrm{exp\_div\_k}'$ with the same equations and collision statements as $\mathrm{exp\_div\_k}$.

This model is included as a macro in CryptoVerif's library of cryptographic primitives, so that it can easily be reused. Similar models also apply to other curves that have a similar structure, for instance Curve448 [LHT16], which

[LHT16] Langley et al., *Elliptic Curves for Security*

is also used by the Noise framework, and by other protocols like TLS 1.3. For Curve448, we have $p = 2^{448} - 2^{224} - 1$, $k = k' = 4$, $q = 2^{446} - 2^{223} - \delta$ with $0 < \delta < 2^{220}$, $q' = 2^{446} + \delta$, and the private keys are $kn$ for $n \in \{2^{445}, \ldots, 2^{446} - 1\}$. We note the following differences with respect to Curve25519:

- Assumptions 1 to 8 are satisfied except that the elements of $\{2^{445}, \ldots, 2^{446} - 1\}$ are not all prime to $qq'$: $q$ is in $\{2^{445}, \ldots, 2^{446} - 1\}$, so $kq$ is a valid private key, and it is the only private key non-prime to $qq'$. It is a weak private key in the sense that for all $X$ that correspond to a point on the curve, $(kq) \cdot X = 0$. Shared secrets generated using this private key are rejected when the participants verify that the shared secrets are non-zero. We first exclude this private key, which yields a probability difference of $1/2^{445} = 2^{-445}$ for each chosen private key, and then apply the previous results with $S = \{2^{445}, \ldots, 2^{446} - 1\} \setminus \{q\}$.
- $q \equiv -1 \mod 4$, so $-1$ is not a square modulo $q$. Hence, in the proof of (3.8), the equalities $x^2 \equiv z \mod q$ and $x^2 \equiv -z \mod q$ cannot both have solutions for $x$ for the same value of $z$. (Either $z$ or $-z$ is a square modulo $q$ but not both.) Therefore, statement (3.8) can be strengthened by reducing the probability difference to $2 \times \mathrm{Pcoll1rand}(Z)$ instead of $4 \times \mathrm{Pcoll1rand}(Z)$.

We have also added to CryptoVerif's library of primitives the model for Curve448 outlined above, as well as a more general model that assumes neither that the private keys are prime to $qq'$ (eliminating weak private keys multiple of $q$ or $q'$ as explained for Curve448), nor that $q \equiv -1 \mod 4$. The latter model is sound for both Curve25519 and Curve448.

We assume that $G_{\mathrm{sub}}$ satisfies the gap Diffie-Hellman (GDH) assumption [OP01], in the following sense:

[OP01] Okamoto and Pointcheval, "The Gap-Problems: a New Class of Problems for the Security of Cryptographic Schemes"

**Definition 3.1** (Gap Diffie-Hellman)**.** Given $g$ that corresponds to a generator of the subgroup of order $q$ of the curve and $\mathcal{E}_U = \{kn \mid n \in [(q+1)/2, q-1]\}$ considered modulo $kqq'$, $a \cdot g$, and $b \cdot g$ for random $a, b \in \mathcal{E}_U$, the adversary has a negligible probability to compute $(ab) \cdot g$ (computational Diffie-Hellman assumption), even when the adversary has access to decisional Diffie-Hellman oracles, which tell it

- given $U, V, W \in G_{\mathrm{sub}}$, whether there exist $\beta \in \mathcal{E}_U$ such that $U = \beta \cdot g$, and $W = \beta \cdot V$;
- given $G, U, V, W \in G_{\mathrm{sub}}$, whether there exist $\alpha, \beta \in \mathcal{E}_U$ such that $G = \alpha \cdot g$, $U = \beta \cdot g$, and $\alpha \cdot W = \beta \cdot V$.

Choosing the exponents in $\mathcal{E}_U$ guarantees the unicity of $\beta$ such that $U = \beta \cdot g$ ($\mathcal{E}$ stands for exponent, $U$ for unique). Moreover, $\mathcal{E}_U$ is close to the set of honestly generated exponents $\{kn \mid n \in S\}$: the statistical distance between the uniform distributions on these two sets is bounded by $2^{-126}$ (see Lemma 4.1). The first decisional Diffie-Hellman oracle basically tells whether $(U, V, W)$ is a good Diffie-Hellman triple with generator $g$, and the second one basically tells whether $(U, V, W)$ is a good Diffie-Hellman triple with generator $G$, as we explain below. We need to separate the two oracles because there exists no $\alpha \in \mathcal{E}_U$ such that $\alpha \cdot W = W$ for all $W \in G_{\mathrm{sub}}$. We use the formulation of Definition 3.1 for the decisional Diffie-Hellman oracles to

accommodate elements $V$, $W$ not necessarily in the group generated by $g$. When all elements are in the group generated by $g$ of order $q$, we have for the first oracle $U = u \cdot g$ for $u = \beta$, $V = v \cdot g$ for some $v$ so $W = (uv) \cdot g$; and for the second oracle $U = u \cdot G$ for $u = \beta/\alpha$ (since $\alpha$ is invertible modulo $q$) and $V = v \cdot G$ for some $v$, so $\alpha \cdot W = \beta \cdot V = (v\beta) \cdot G = (uv\alpha) \cdot G$ so $W = (uv) \cdot G$, hence there exist $u$ and $v$ such that $U = u \cdot G$, $V = v \cdot G$, and $W = (uv) \cdot G$. $(U, V, W)$ is then a good Diffie-Hellman triple with generator $G$. Note that we need the computational Diffie-Hellman assumption only for the curve, not for the twist, while the decisional Diffie-Hellman oracles may mix elements of the curve and of the twist: $G$ and $U$ always correspond to points of the curve, while $V$ and $W$ either correspond to points both on the curve or both on the twist. (If $V$ corresponds to a point on the curve and $W$ corresponds to a point on the twist, then $\beta \cdot V$ corresponds to a point on the curve and $\alpha \cdot W$ corresponds to a point on the twist, so they cannot be equal except when $V = W = 0$.) This assumption was already modelled in CryptoVerif.

Interestingly, the obtained model is very similar to what we would obtain with the elliptic curve Curve25519 itself, without using a single coordinate ladder. In the latter case, $G$ is the curve itself, a group of order $kq$. The base point $g$ has prime order $q$; $G_{\mathrm{sub}}$ is the prime order subgroup generated by $g$. The set $Z$ is the set of integers multiple of $k$ and prime to $q$, modulo $kq$. The operation $y \cdot X$ is point multiplication, the functions decode_pk and encode_pk are the identity. Then we have properties similar to Lemma 3.1, replacing $qq'$ with $q$: $y \cdot X$ is defined for $y \in \mathbb{Z}_{kq}$; $y \cdot (z \cdot X) = (yz) \cdot X$; for any $z \in Z$, $X, Y \in G$, we have $z \cdot X = z \cdot Y$ if and only if $k \cdot X = k \cdot Y$ (because $z = kz'$ for some $z'$ invertible modulo $q$); for any $z$ prime to $q$, for any $X, Y \in G_{\mathrm{sub}}$, we have $z \cdot X = z \cdot Y$ if and only if $X = Y$ (because $z$ is invertible modulo $q$). Public keys $X$ and $Y$ are equivalent when $k \cdot X = k \cdot Y$, so each public key has $k$ equivalent public keys, including itself. Similarly to the previous model, $\exp(X, y) = y \cdot X$, $\mathrm{mult}$ is the product modulo $kq$ in $Z$, $\mathrm{pow\_k}(X) = k \cdot X$, and $\mathrm{exp\_div\_k}(X, y) = (y/k) \cdot X$. The function $\mathrm{G_{sub}2G}$ maps each element of $G_{\mathrm{sub}}$ to the same element in $G$. The constants $\mathrm{zero}$ in $G$ and $\mathrm{zero_{sub}}$ in $G_{\mathrm{sub}}$ represent the neutral element of $G$, that is, the point at infinity $\infty$. The properties of these functions and constants until (3.4) are proved similarly to the previous model, replacing $qq'$ with $q$. Let us solve the equation $y \cdot X = z \cdot X$ for $X \in G_{\mathrm{sub}}$. If $X = \infty$, then $y \cdot X = \infty = z \cdot X$ for all $y$ and $z$. Otherwise, $X$ is a generator of $G_{\mathrm{sub}}$, so $y \cdot X = z \cdot X$ if and only if $y \equiv z \mod q$. As a result, we can strengthen (3.5) as follows:

$$\mathrm{equation\ forall}\ X : G_{\mathrm{sub}}, y : Z, z : Z;$$
$$(\mathrm{exp\_div\_k}(X, y) = \mathrm{exp\_div\_k}(X, z)) = ((y = z) \vee (X = \mathrm{zero_{sub}})).$$

In the proof of (3.6), $(x/k) \cdot X = n \cdot X = Y = y \cdot X$ holds if and only if $n \equiv y \mod q$, and the probability that the expressions differ is at most Pcoll1rand($Z$) instead of $2 \times$ Pcoll1rand($Z$). Due to the strengthened form of (3.5), equalities $\mathrm{exp\_div\_k}(\mathrm{g\_k}, y) = \mathrm{exp\_div\_k}(\mathrm{g\_k}, z)$ are replaced with $y = z$. For this reason, we replace (3.7), (3.8), and (3.9) with properties

on private keys. Statement (3.7) becomes:

$$\text{collision } x \xleftarrow{R} Z; \text{forall } y : Z, z : Z;$$
$$\text{return}(\text{mult}(x, y) = z) \approx_{\text{Pcoll1rand}(Z)} \text{return}(\text{false})$$
$$\text{if } y \text{ independent-of } x \wedge z \text{ independent-of } x$$

since $\text{mult}(x, y) = z$ means $xy \equiv z \mod kq$, so $xy/k \equiv z/k \mod q$ and $y$ is invertible modulo $q$, so $x/k \equiv z/ky \mod q$, so $x \equiv z/y \mod kq$. Since $z/y$ is independent of $x$, the equality $x \equiv z/y \mod kq$ has probability at most Pcoll1rand$(Z)$ to happen. Statement (3.8) becomes:

$$\text{collision } x \xleftarrow{R} Z; y \xleftarrow{R} Z; [\text{random\_choices\_may\_be\_equal}] \text{ forall } z : Z;$$
$$\text{return}(\text{mult}(x, y) = z) \approx_{2\times\text{Pcoll1rand}(Z)} \text{return}(\text{false})$$
$$\text{if } z \text{ independent-of } x \vee z \text{ independent-of } y.$$

When $x$ and $y$ are independent, this statement is a consequence of the previous one. When they are the same random choice, $x = y$, and $\text{mult}(x, x) = z$ implies $x^2 \equiv z \mod kq$, so $x^2/k \equiv z/k \mod q$ so $(x/k)^2 \equiv z/k^2 \mod q$ since $k$ is invertible modulo $q$, and the probability that the expressions differ is at most $2 \times$ Pcoll1rand$(Z)$ instead of $4 \times$ Pcoll1rand$(Z)$ since each element has at most 2 square roots modulo $q$. The increased probability that we observe in the model with single coordinate ladder in the previous collision statements comes from the existence of equivalent private keys. Statement (3.9) becomes

$$\text{equation forall } x : Z, y : Z, y' : Z; (\text{mult}(x, y) = \text{mult}(x, y')) = (y = y')$$

since $\text{mult}(x, y) = \text{mult}(x, y')$ means $xy \equiv xy' \mod kq$, so $xy/k \equiv xy'/k \mod q$, so $y/k \equiv y'/k \mod q$ since $x$ is invertible modulo $q$, so $y \equiv y' \mod kq$, that is, $y = y'$ in $Z$. Finally, the GDH assumption is needed only in the subgroup $G_{\text{sub}}$, so the twist is excluded. This model is presented in the conference version of this work [LBB19a], where we unfortunately overlooked that Curve25519 implementations use a single coordinate ladder. To sum up, the main difference with the model using a single coordinate ladder is that some collisions have a slightly higher probability with a single coordinate ladder. Hence, in practice, using a single coordinate ladder has little impact on security.

[LBB19a] Lipp et al., "A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol"

In their cryptographic proof of WireGuard, Dowling and Paterson [DP18] use the PRF-ODH assumption. We use the GDH and random oracle assumptions instead because CryptoVerif cannot currently use the PRF-ODH assumption in scenarios with key compromise. While in principle the PRF-ODH assumption is weaker, Brendel, Fischlin, Günther, and Janson [Bre+17] show that it is implausible to instantiate the PRF-ODH assumption without a random oracle, so our assumptions and the one of [DP18] are in fact fairly similar.

[DP18] Dowling and Paterson, "A Cryptographic Analysis of the WireGuard Protocol"

[Bre+17] Brendel et al., "PRF-ODH: Relations, Instantiations, and Impossibility Results"

## 3.4 Indifferentiability of Hash Chains

Before modelling WireGuard, we first present a different, equally precise, formulation of hash chains that is more amenable to a mechanised proof in CryptoVerif. Indeed, WireGuard makes many hash oracle calls to BLAKE2s,

and at each call to a random oracle, CryptoVerif tests whether the arguments are the same as in any other previous random oracle call (to return the previous result of the random oracle). Therefore, using directly BLAKE2s as a random oracle would introduce a very large number of cases and yield exaggeratedly large cryptographic games. In order to avoid that, we simplify the random oracle calls using indifferentiability lemmas. These lemmas are not specific to WireGuard and can be used to simplify sequences of random oracle calls in other protocols, including other Noise protocols and Signal [MP16]. In the future, these lemmas may serve as a basis for an indifferentiability prover inside CryptoVerif, which would simplify random oracle calls before proving the protocol.

[MP16] Marlinspike and Perrin, *The X3DH Key Agreement Protocol*

Specifically, WireGuard uses HKDF in a chain of calls to derive symmetric keys at different stages of the protocol:

$$\begin{aligned}
&C_0 && \leftarrow \mathsf{const}\\
&C_1 && \leftarrow \mathsf{hkdf}_1(C_0, v_0)\\
&C_2 \| k_1 && \leftarrow \mathsf{hkdf}_2(C_1, v_1)\\
&C_3 \| k_2 && \leftarrow \mathsf{hkdf}_2(C_2, v_2)\\
&C_4 && \leftarrow \mathsf{hkdf}_1(C_3, v_3)\\
&C_5 && \leftarrow \mathsf{hkdf}_1(C_4, v_4)\\
&C_6 && \leftarrow \mathsf{hkdf}_1(C_5, v_5)\\
&C_7 \| \pi \| k_3 && \leftarrow \mathsf{hkdf}_3(C_6, v_6)\\
&T^{\rightarrow} \| T^{\leftarrow} && \leftarrow \mathsf{hkdf}_2(C_7, v_7)
\end{aligned}$$

We show, using the indifferentiability lemmas of this section, that $\mathsf{hkdf}_n$ is indifferentiable from a random oracle, and that the chain above is indifferentiable from:

$$\begin{aligned}
&k_1 && \leftarrow \mathsf{chain}'_1(v_0, v_1)\\
&k_2 && \leftarrow \mathsf{chain}'_2(v_0, v_1, v_2)\\
&\pi \| k_3 \| T^{\rightarrow} \| T^{\leftarrow} && \leftarrow \mathsf{chain}'_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6)
\end{aligned} \tag{3.10}$$

where $\mathsf{chain}'_1$, $\mathsf{chain}'_2$, and $\mathsf{chain}'_6$ are independent random oracles. Thus, we obtain a much simpler computation, which we use in our CryptoVerif model of WireGuard. Previous analyses of WireGuard did not use such a result because they do not rely on the random oracle model: [DP18] relies on the PRF-ODH assumption, [DM18] uses the symbolic model.

[DP18] Dowling and Paterson, "A Cryptographic Analysis of the WireGuard Protocol"

[DM18] Donenfeld and Milner, *Formal Verification of the WireGuard Protocol*

### 3.4.1 *Definition of Indifferentiability*

Indifferentiability can be defined as follows. This definition is an extension of [Cor+05] to several independent oracles.

[Cor+05] Coron et al., "Merkle-Damgård Revisited: How to Construct a Hash Function"

**Definition 3.2** (Indifferentiability)**.** Functions $(F_i)_{1\leq i\leq n}$ with oracle access to independent random oracles $(H_j)_{1\leq j\leq m}$ are $(t_D, t_S, (q_{H_j})_{1\leq j\leq m}, (q_{F_i})_{1\leq i\leq n}, (q_{H'_i})_{1\leq i\leq n}, \epsilon)$-indifferentiable from independent random oracles $(H'_i)_{1\leq i\leq n}$ if there exists a simulator $S$ such that for any distinguisher $D$

$$|\Pr[D^{(F_i)_{1\leq i\leq n},(H_j)_{1\leq j\leq m}} = 1] - \Pr[D^{(H'_i)_{1\leq i\leq n},S} = 1]| \leq \epsilon$$

The simulator $S$ has oracle access to $(H'_i)_{1\le i\le n}$, makes at most $q_{H'_i}$ queries to $H'_i$, and runs in time $t_S$. The distinguisher $D$ runs in time $t_D$ and makes at most $q_{H_j}$ queries to $H_j$ for $1 \le j \le m$ and $q_{F_i}$ queries to $F_i$ for $1 \le i \le n$.

In the game $G_0 = D^{(F_i)_{1\le i\le n},(H_j)_{1\le j\le m}}$, the distinguisher interacts with the real functions $F_i$ and the random oracles $H_j$ from which the functions $F_i$ are defined. In the game $G_1 = D^{(H'_i)_{1\le i\le n},S}$, the distinguisher interacts with independent random oracles $H'_i$ instead of $F_i$, and with a simulator $S$, which simulates the behaviour of the random oracles $H_j$ using calls to $H'_i$. (We may also present $S$ as $m$ simulators $S_j$ that each simulate a single random oracle $H_j$ using calls to $H'_i$, $1 \le i \le n$; these simulators share a common state.) Indifferentiability means that these two games are indistinguishable.

### 3.4.2 *Basic Lemmas*

In this section, we show several basic indifferentiability lemmas, which are not specific to WireGuard. The proofs that are not included in this section can be found in the appendix. Lemma 3.2 shows that random oracle calls with disjoint domains are indifferentiable from calls to independent random oracles.

**Lemma 3.2** (Version of [KBB17, Lemma 2] with more precise evaluation of numbers of oracle calls)**.** *If $H$ is a random oracle, then the functions $H_1, \dots, H_n$ defined as $H$ on disjoint subsets $D_1, \dots, D_n$ of the domain $D$ of $H$ are $(t_D, t_S, q_H, (q_{H_i})_{1\le i\le n}, (q'_{H_i})_{1\le i\le n}, 0)$-indifferentiable from independent random oracles, where $t_S = \mathcal{O}(q_H)$ assuming one can determine in constant time to which subset $D_i$ an element belongs, and $q'_{H_i}$ is the number of requests to $H$ in domain $D_i$ made by the distinguisher. Hence $q'_{H_1} + \dots + q'_{H_n} \le q_H$, so in the worst case $q'_{H_i}$ is bounded by $q_H$.*

Lemma 3.3 shows that the concatenation of two independent random oracle calls is indifferentiable from a random oracle.

**Lemma 3.3.** *If $H_1$ and $H_2$ are independent random oracles with the same domain that return bitstrings of length $l_1$ and $l_2$ respectively, then the concatenation $H'$ of $H_1$ and $H_2$ is $(t_D, t_S, (q_{H_1}, q_{H_2}), q_{H'}, q_{H_1} + q_{H_2}, 0)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q_{H_1} + q_{H_2})$.*

Conversely, Lemma 3.4 shows that splitting the output of a random oracle into two fixed length outputs yields independent random oracles.

**Lemma 3.4.** *If $H$ is a random oracle that returns bitstrings of length $l$, then the function $H'_1$ returning the first $l_1$ bits of $H$ and the function $H'_2$ returning the last $l - l_1$ bits of $H$ are $(t_D, t_S, q_H, (q_{H'_1}, q_{H'_2}), (q_H, q_H), 0)$-indifferentiable from independent random oracles, where $t_S = \mathcal{O}(q_H)$.*

As a particular consequence, Lemma 3.5 shows that the truncation of a random oracle is indifferentiable from a random oracle.

**Lemma 3.5** (Already stated in [KBB17, Lemma 3])**.** *If $H$ is a random oracle that returns bitstrings of length $l$, then the truncation $H'$ of $H$ to length $l' < l$ is $(t_D, t_S, q_H, q_{H'}, q_H, 0)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q_H)$.*

Lemmas 3.6 and 3.7 deal with the composition of two random oracle calls in sequence. We extended CryptoVerif to be able to prove indistinguishability between two games given by the user. Thanks to this extension, CryptoVerif helps considerably with the proof of these lemmas: it shows the indistinguishability result between the games $G_0$ and $G_1$ described in Section 3.4.1, which implies the indifferentiability result. We present the proof of Lemma 3.6 as an illustration.

**Lemma 3.6.** *If $H_1 : S_1 \to S_1'$ and $H_2 : S_1' \times S_2 \to S_2'$ are independent random oracles, then $H_3$ defined by $H_3(x,y) = H_2(H_1(x),y)$ is $(t_D, t_S, (q_{H_1}, q_{H_2}), q_{H_3}, q_{H_2}, \epsilon)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q_{H_1} q_{H_2})$ and $\epsilon = (2q_{H_2}q_{H_1} + q_{H_1}^2 + q_{H_2}q_{H_3} + q_{H_3}^2)/|S_1'|$.*

*Proof.* Consider

- the game $G_0$ in which $H_1$ and $H_2$ are independent random oracles, and $H_3(x,y) = H_2(H_1(x),y)$, and
- the game $G_1$ in which $H_3$ is a random oracle; two lists $L_1$ and $L_2$ that are initially empty; $H_1(x)$ returns $y$ if $(x,y) \in L_1$ for some $y$, and otherwise chooses a fresh random $r$ in $S_1'$, adds $(x,r)$ to $L$ and returns $r$; $H_2(y,z)$ returns $H_3(x,z)$ if $(x,y) \in L_1$ for some $x$, otherwise returns $u$ if $((y,z),u) \in L_2$ for some $u$, and otherwise chooses a fresh random $r$ in $S_2'$, adds $((y,z),r)$ to $L_2$ and returns $r$.

CryptoVerif shows that the games $G_0$ and $G_1$ are indistinguishable, up to probability $\epsilon$. □

**Lemma 3.7.** *If $H_1 : S_1 \to S_1'$ and $H_2 : S_1' \times S_1 \to S_2'$ are independent random oracles, then $H_1' = H_1$ and $H_2'$ defined by $H_2'(x) = H_2(H_1(x),x)$ are $(t_D, t_S, (q_{H_1}, q_{H_2}), (q_{H_1'}, q_{H_2'}), (q_{H_1} + q_{H_2}, q_{H_2}), \epsilon)$-indifferentiable from independent random oracles, where $t_S = \mathcal{O}(q_{H_2})$ and $\epsilon = q_{H_2}(2q_{H_1} + 2q_{H_1'} + q_{H_2'} + 1)/|S_1'|$.*

### 3.4.3 *Indifferentiability of HKDF*

The hkdf key derivation function is defined as follows [KE10]:

[KE10] Krawczyk and Eronen, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*

$$\mathsf{hkdf\text{-}extract}(\mathrm{salt}, \mathrm{key}) = \mathsf{hmac}(\mathrm{salt}, \mathrm{key})$$

$$\mathsf{hkdf\text{-}expand}_n(\mathrm{prk}, \mathrm{info}) = k_1 \| \ldots \| k_n \text{ where}$$

$$k_1 = \mathsf{hmac}(\mathrm{prk}, \mathrm{info} \| i_0)$$

$$k_{i+1} = \mathsf{hmac}(\mathrm{prk}, k_i \| \mathrm{info} \| i + i_0) \text{ for } 1 \le i < n$$

$$\mathsf{hkdf}_n(\mathrm{salt}, \mathrm{key}, \mathrm{info}) = \mathsf{hkdf\text{-}expand}_n(\mathsf{hkdf\text{-}extract}(\mathrm{salt}, \mathrm{key}), \mathrm{info})$$

where $n \le 255$, and $i_0 = 0x01$ and $i$ are of size 1 byte. In WireGuard, info is always empty, so we omit it in Section 3.2. Let $\mathcal{S}$, $\mathcal{K}$, and $\mathcal{I}$ be the sets of possible values of salt, key, and info respectively, and $\mathcal{M}$ the output of hmac.

We suppose that hmac is a random oracle, and we show that $\mathsf{hkdf\text{-}expand}_n$ is indifferentiable from a random oracle.

**Lemma 3.8.** *If* hmac *is a random oracle, then* $\mathsf{hkdf\text{-}expand}_n$ *is $(t_D, t_S, q_{\mathsf{hmac}}, q_{\mathsf{hkdf\text{-}expand}_n}, q_{\mathsf{hmac}}, \epsilon)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q_{\mathsf{hmac}})$ and $\epsilon = (3q_{\mathsf{hkdf\text{-}expand}_n} q_{\mathsf{hmac}} + q_{\mathsf{hmac}}^2)/|\mathcal{M}|$.*

Using this result, we show that $\mathsf{hkdf}_n$ is indifferentiable from a random oracle, with the additional assumption that the calls to hmac use disjoint domains. (We show that this assumption is necessary and give full proofs of these results in Appendix B.)

**Lemma 3.9.** *If* hmac *is a random oracle and* $\mathcal{K} \cap (\mathcal{I}\|i_0 \cup \bigcup_{i=1}^{n-1} \mathcal{M}\|\mathcal{I}\|i + i_0) = \emptyset$*, then* $\mathsf{hkdf}_n$ *with domain* $\mathcal{S} \times \mathcal{K} \times \mathcal{I}$ *is* $(t_D, t_S, q_{\mathsf{hmac}}, q_{\mathsf{hkdf}_n}, q_{\mathsf{hmac}}, \epsilon)$*-indifferentiable from a random oracle, where* $t_S = \mathcal{O}(q_{\mathsf{hmac}}^2)$ *and* $\epsilon = (q_{\mathsf{hkdf}_n}^2 + 4q_{\mathsf{hkdf}_n} q_{\mathsf{hmac}} + q_{\mathsf{hmac}}^2)/|\mathcal{M}|$.

This result extends the proof given for $\mathsf{hkdf}_2$ in [KBB17, Lemma 1]. Moreover, our proof is modular and partly made using CryptoVerif, thanks to the basic lemmas of Section 3.4.2.

[KBB17] Kobeissi et al., "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach"

*Proof sketch.* Since the domains are disjoint, by Lemma 3.2, the $(n+1)$ calls to hmac are indifferentiable from independent random oracles $H_0, \ldots, H_n$. The constant $i + i_0$ can be removed from the arguments of $H_{i+1}$ since it is fixed for a given $H_{i+1}$. By Lemma 3.7, the computation of $k_2 = H_2(H_1(\mathrm{prk}, \mathrm{info}), \mathrm{prk}, \mathrm{info})$ is indifferentiable from a random oracle $k_2 = H_2'(\mathrm{prk}, \mathrm{info})$. Applying this reasoning $n$ times, the computation of $k_i$ for $1 \leq i \leq n$ is indifferentiable from independent random oracles $k_i = H_i'(\mathrm{prk}, \mathrm{info})$. By Lemma 3.3, concatenation of $H_i'$ for $1 \leq i \leq n$ is indifferentiable from a random oracle $H$, so $\mathsf{hkdf}_n(\mathrm{salt}, \mathrm{key}, \mathrm{info}) = k_1\| \ldots \|k_n = H(\mathrm{prk}, \mathrm{info})$, where $\mathrm{prk} = H_0(\mathrm{salt}, \mathrm{key})$. By Lemma 3.6, we conclude that $\mathsf{hkdf}_n$ is indifferentiable from a random oracle. □

### 3.4.4 *Indifferentiability of a Chain of Random Oracle Calls*

In this section, we prove the indifferentiability of a chain of random oracle calls defined as follows.

**Definition 3.3** (Chain)**.** Let $m \geq 1$ be a fixed integer, let $C$ and $C_j$ with $0 \leq j \leq m+1$ be bitstrings of length $l'$, let $v_j$ with $0 \leq j \leq m$ be bitstrings of arbitrary length, let $l$ be the length of the output of $\mathsf{H}(C_j, v_j)$, and let $r_j$ with $0 \leq j \leq m$ be bitstrings of length $(l - l')$. We define the functions $\mathsf{chain}_n, 0 \leq n < m$ and the function $\mathsf{chain}_m$ in the following way:

$$\begin{aligned} &\mathsf{chain}_n(v_0, \ldots, v_n) = \\ &\quad C_0 = \mathsf{const} \\ &\quad \text{for } j = 0 \text{ to } n \text{ do } C_{j+1}\|r_j = \mathsf{H}(C_j, v_j) \\ &\quad \text{return } r_n \end{aligned} \tag{3.11}$$

$$\begin{aligned} &\mathsf{chain}_m(v_0, \ldots, v_m) = \\ &\quad C_0 = \mathsf{const} \\ &\quad \text{for } j = 0 \text{ to } m \text{ do } C_{j+1}\|r_j = \mathsf{H}(C_j, v_j) \\ &\quad \text{return } C_{m+1}\|r_m \end{aligned} \tag{3.12}$$

The functions $\mathsf{chain}_n, n < m$, have an output of length $(l - l')$, and the output length of $\mathsf{chain}_m$ is $l$.

**Lemma 3.10.** *If* $\mathsf{H}$ *is a random oracle, then* $\mathsf{chain}_n$*, for* $n \leq m$*, are* $(t_D, t_S, q_{\mathsf{H}}, (q_{\mathsf{chain}_n})_{0 \leq n \leq m}, (q_{\mathsf{H}})_{0 \leq n \leq m}, \epsilon)$*-indifferentiable from independent random oracles, where* $t_S = \mathcal{O}(q_{\mathsf{H}}^2)$ *and* $\epsilon = \left(\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n}\right) \cdot q_{\mathsf{H}} + q_{\mathsf{H}}^2\right)/2^{l'}$.

This lemma is proved in the appendix. We could probably prove it for small values of $m$ using CryptoVerif, but the generic result requires a manual proof because CryptoVerif does not support loops.

### 3.4.5 *Application to WireGuard*

WireGuard employs BLAKE2s [Aum+13] both directly as the function hash and indirectly as hash function in hmac and thus also in hkdf. In our proof, we assume that hash is collision-resistant and use the random oracle assumption for usages of BLAKE2s via hkdf. Rigorously, to be able to use two distinct assumptions, we need the domains of these two uses to be disjoint. This is true in WireGuard: the length of the argument of hash is 64 bytes for $\mathsf{hash}(H_0 \| S_r^{\mathrm{pub}})$, $\mathsf{hash}(H_1 \| E_i^{\mathrm{pub}})$, $\mathsf{hash}(H_4 \| E_r^{\mathrm{pub}})$, and $\mathsf{hash}(H_5 \| \pi)$, 80 bytes for $\mathsf{hash}(H_2 \| S_{i\,\text{🔒}}^{\mathrm{pub}})$, 60 bytes for $\mathsf{hash}(H_3 \| ts_{\text{🔒}})$, 48 bytes for $\mathsf{hash}(H_6 \| \mathrm{empty}_{\text{🔒}})$, 40 bytes for $\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_r^{\mathrm{pub}})$, $\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_i^{\mathrm{pub}})$, and $\mathsf{hash}(\mathsf{label}_{\mathsf{cookie}} \| S_m^{\mathrm{pub}})$. In contrast, the length of the argument of BLAKE2s in $\mathsf{hmac}(k, m)$ is 96 bytes or $64 + \mathsf{length}(m)$, and in the computation of hkdf, $\mathrm{info}$ is empty in WireGuard, so the length of $m$ is 32 bytes $(\mathrm{key})$, 1 byte $(\mathrm{info} \| i_0)$ or 33 bytes $(k_i \| \mathrm{info} \| i + i_0)$, so the length of the argument of BLAKE2s in the computation of hkdf is 96, 65, or 97 bytes.

[Aum+13] Aumasson et al., "BLAKE2: Simpler, Smaller, Fast as MD5"

Then by Lemma 3.2, we can consider two independent random oracles, hash for the direct uses and $\mathsf{hash}'$ for the uses via hkdf. Since hash is a random oracle, it is a fortiori collision-resistant.

Since $\mathsf{hash}'$ is a random oracle, hmac-$\mathsf{hash}'$ is indifferentiable from a random oracle by [Dod+12, Theorem 3].

[Dod+12] Dodis et al., "To Hash or Not to Hash Again? (In)Differentiability Results for $H^2$ and HMAC"

Moreover, the domains of the calls to hmac in $\mathsf{hkdf}_n$ are disjoint. Indeed, $\mathcal{K}$ consists of bitstrings of length 32 bytes, $\mathcal{I} \| i_0$ consists of bitstrings of length 1 byte, and $\mathcal{M} \| \mathcal{I} \| i + i_0$ consists of bitstrings of length 33 bytes. By Lemma 3.9, $\mathsf{hkdf}_n$ is indifferentiable from a random oracle.

In this section, we use the $\|$ operator to concatenate blocks of 32 bytes and the placeholder _ for one unnamed 32-byte block. Since we prove Lemma 3.10 for a chain of calls to the same $\mathsf{hkdf}_n$ function, we rewrite the chain of hkdf calls in WireGuard to use only calls to $\mathsf{hkdf}_3$, as 3 is the maximum number of outputs needed:

$$
\begin{aligned}
&C_0 && \leftarrow \mathsf{const}\\
&C_1 \| \_ \| \_ && \leftarrow \mathsf{hkdf}_3(C_0, v_0)\\
&C_2 \| k_1 \| \_ && \leftarrow \mathsf{hkdf}_3(C_1, v_1)\\
&C_3 \| k_2 \| \_ && \leftarrow \mathsf{hkdf}_3(C_2, v_2)\\
&C_4 \| \_ \| \_ && \leftarrow \mathsf{hkdf}_3(C_3, v_3)\\
&C_5 \| \_ \| \_ && \leftarrow \mathsf{hkdf}_3(C_4, v_4)\\
&C_6 \| \_ \| \_ && \leftarrow \mathsf{hkdf}_3(C_5, v_5)\\
&C_7 \| \pi \| k_3 && \leftarrow \mathsf{hkdf}_3(C_6, v_6)\\
&T^{\rightarrow} \| T^{\leftarrow} \| \_ && \leftarrow \mathsf{hkdf}_3(C_7, v_7)
\end{aligned}
$$

Because of the way $\mathsf{hkdf}_n$ is constructed, this is actually the same computation.

By Lemma 3.10, the computation above can be replaced with the following one:

$$\begin{aligned}
&_\|_ && \leftarrow \mathsf{chain}_0(v_0)\\
&k_1\|_ && \leftarrow \mathsf{chain}_1(v_0, v_1)\\
&k_2\|_ && \leftarrow \mathsf{chain}_2(v_0, v_1, v_2)\\
&_\|_ && \leftarrow \mathsf{chain}_3(v_0, v_1, v_2, v_3)\\
&_\|_ && \leftarrow \mathsf{chain}_4(v_0, v_1, v_2, v_3, v_4)\\
&_\|_ && \leftarrow \mathsf{chain}_5(v_0, v_1, v_2, v_3, v_4, v_5)\\
&\pi\|k_3 && \leftarrow \mathsf{chain}_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6)\\
&T^{\rightarrow}\|T^{\leftarrow}\|_ && \leftarrow \mathsf{chain}_7(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)
\end{aligned}$$

where $\mathsf{chain}_i$ for $i \leq 7$ are independent random oracles.

The output of the random oracles can be truncated by Lemma 3.5 to avoid having to throw away parts of the output:

$$\begin{aligned}
&k_1 && \leftarrow \mathsf{chain}'_1(v_0, v_1)\\
&k_2 && \leftarrow \mathsf{chain}'_2(v_0, v_1, v_2)\\
&\pi\|k_3 && \leftarrow \mathsf{chain}_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6)\\
&T^{\rightarrow}\|T^{\leftarrow} && \leftarrow \mathsf{chain}'_7(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)
\end{aligned}$$

where $\mathsf{chain}'_1$, $\mathsf{chain}'_2$, $\mathsf{chain}_6$, and $\mathsf{chain}'_7$ are independent random oracles. In WireGuard, $v_7 = \mathrm{empty}$, so $T^{\rightarrow}$ and $T^{\leftarrow}$ only depend on $v_0, \ldots, v_6$, as do $\pi$ and $k_3$ in the previous line. By Lemma 3.3, we can replace the last two lines with one random oracle call:

$$\begin{aligned}
&k_1 && \leftarrow \mathsf{chain}'_1(v_0, v_1)\\
&k_2 && \leftarrow \mathsf{chain}'_2(v_0, v_1, v_2)\\
&\pi\|k_3\|T^{\rightarrow}\|T^{\leftarrow} && \leftarrow \mathsf{chain}'_6(v_0, v_1, v_2, v_3, v_4, v_5, v_6)
\end{aligned}$$

where $\mathsf{chain}'_1$, $\mathsf{chain}'_2$, and $\mathsf{chain}'_6$ are independent random oracles.

## 3.5 Modelling WireGuard

This section presents our model of the WireGuard protocol in CryptoVerif. We prove security properties for that model in Section 3.6.

### 3.5.1 *Execution Environment*

In our model, we consider two honest entities $A$ and $B$. In the initial setup, we generate the static key pairs for these two entities and publish their public keys, so that the adversary can use them. After this setup, we run parallel processes that represent a number of executions of $A$ and $B$ polynomial in the security parameter.

The entities $A$ and $B$ can play both the initiator and responder role. These two entities can run WireGuard between each other, but also with any number of dishonest entities included in the adversary: for each session, the adversary sends to the initiator its *partner public key*, that is, the public key

of the entity with which it should start a session; the adversary sends to the responder the set of partner public keys that it accepts messages from.

This setting allows us to prove security for any sessions between two honest entities, in a system that may contain any number of (honest or dishonest) other entities. We prove security for sessions in which $A$ is the initiator and $B$ is the responder. We do not explicitly prove security for sessions in which $B$ is the initiator and $A$ is the responder, but the same security properties hold by symmetry.

The processes for the entities $A$ and $B$ model the entire protocol, including the first two protocol messages, the key confirmation message from the initiator, and then a number of transport data messages polynomial in the security parameter, in both directions between initiator and responder. The model also includes random oracles, and we allow the adversary to call any of the random oracles that we use.

We consider 3 variants of this model:

VARIANT 1. This variant does not rely at all on the pre-shared key for proving security, so $A$ and $B$ receive a pre-shared key chosen by the adversary at the beginning of each execution. That allows the adversary to model both the absence of a pre-shared key (by choosing the value 0) or a compromised pre-shared key of its choice.

We model the dynamic compromise of the private static key of $A$ (resp. $B$) by a process that the adversary can call at any time and that returns the private key of $A$ (resp. $B$) and records the compromise by defining a particular variable, so that it can be tested in the security properties that we consider.

In WireGuard, four Diffie-Hellman operations and the pre-shared key contribute to the session keys. If the pre-shared key is not used or compromised, security is based on the four Diffie-Hellman operations. If one of them cannot be computed by the adversary, then the session keys are secret. Therefore, we consider all combinations of compromises but those where both keys on one side are compromised, that is:

1. $A$ and $B$'s private static keys may be dynamically compromised;
2. $A$'s private static key may be dynamically compromised and $B$'s private ephemeral key is compromised (by sending it to the adversary as soon as it is chosen);
3. $B$'s private static key may be dynamically compromised and $A$'s private ephemeral key is compromised;
4. $A$ and $B$'s private ephemeral keys are compromised.

We prove most security properties for *clean* sessions, that is, intuitively, sessions between honest entities; cleanliness is the minimal assumption needed to hope for security. A session of $A$ is clean when either $B$'s private static key is not compromised yet and $A$'s partner public key is equivalent to $B$'s static public key, or $B$'s private static key is compromised and the public ephemeral key received by $A$ is equivalent to a non-compromised ephemeral generated by $B$. $B$'s session cleanliness is defined symmetrically. Intuitively, when $B$'s private static key is not compromised, $A$ can rely on that key to authenticate $B$, so $A$ thinks she talks to $B$ when she runs a

session with $B$'s public key. We consider a public key equivalent to $B$'s public key rather than equal to $B$'s public key to strengthen the properties: the authentication property shown in Section 3.6 then implies that when $A$ successfully runs a session with a partner public key equivalent to $B$'s public key, then these two keys are in fact equal. (We find an interesting scenario concerning equivalent public keys and identity mis-binding with variant 3 of our model, we discuss it in Section 3.6.) When $B$'s private static key is compromised, $A$ cannot authenticate $B$, but we can still prove security when the ephemeral key received by $A$ has been generated by $B$. Like for static keys, when $A$ successfully runs a session with a received ephemeral equivalent to an ephemeral generated by $B$, then these two ephemerals are in fact equal. (Instead of considering compromised ephemeral keys, we could also have modelled dishonestly generated ephemeral keys. We expect that some properties shown in Section 3.6, such as session uniqueness, would not hold in this case.)

VARIANT 2. This variant relies exclusively on the pre-shared key for security. In that variant, we consider all private static and ephemeral keys as always compromised. We choose a pre-shared key randomly in the initial setup, and run sessions between $A$ and $B$ with that pre-shared key. In this model, $A$'s partner public key is always $B$'s public key and symmetrically, and these sessions between $A$ and $B$ are always considered clean. The adversary can run $A$'s and $B$'s sessions with other entities since $A$ and $B$'s private static keys are compromised and these sessions use a different pre-shared key.

VARIANT 3. In this variant, all keys are compromised: all private static and ephemeral keys are always compromised and the pre-shared key is chosen by the adversary for each session. This model is useful for proving properties that do not rely on session cleanliness, that is, properties that hold even for sessions involving dishonest participants.

With this model, we analyse the whole WireGuard protocol as it is, tying together the authenticated key exchange and the transport data phase. A similar approach was chosen by the creators of the Authenticated and Confidential Channel Establishment (ACCE) [Jag+12] model to analyse TLS. Instead of reasoning about key indistinguishability, ACCE looks at the security of the messages exchanged encrypted using the key. We do the same, for the key confirmation and all subsequent transport data messages.

[Jag+12] Jager et al., "On the Security of TLS-DHE in the Standard Model"

In ACCE, the adversary has to choose one clean test session in which it tries to break security by determining the secret bit. In all other sessions, it is allowed to reveal the session keys. In our model, all clean sessions are test sessions, and we explicitly reveal the session keys in sessions that are not clean.

### 3.5.2 *Modelling Tricks*

Apart from the HKDF chains where we prove that the way we model them is indifferentiable from the real protocol in Section 3.4, we use the following modelling tricks:

- Timestamps: CryptoVerif has no support for time, so instead of generating the timestamp, we input it from the adversary. In other words, we delegate the task of timestamp generation to the adversary. In order to model replay protection for the first message, the responder stores a global table (that is, a list) of triples containing the received timestamp, the partner public key for that session, as well as its own public key. (This is equivalent to having a distinct table of timestamps and partner public keys for each responder, represented by its public key.) The responder rejects the first message when the triple (received timestamp, partner public key, and responder public key) is already in the table.
- Nonces for the AEAD scheme: The nonces in WireGuard are computed by incrementing a counter. CryptoVerif has no support for that, so we receive the desired value of the counter from the adversary. We guarantee that the same counter is never used twice in the same session for sending messages by storing all counters used for sending messages in a table of pairs (session index, counter), where the session index identifies the session uniquely: it indicates whether $A$ or $B$ is running, as initiator or as responder, and contains a unique integer index for the execution of that entity in that role. This is equivalent to having a distinct table of counters for each session. The message is not sent when the adversary provides a counter that is already in the table. We guarantee that the same counter is never used twice for receiving messages in the same way, using a separate table.
- We omit the MACs $\mathrm{mac}_1$ and $\mathrm{mac}_2$ in our model. This simplifies the proof but preserves its soundness, since they can be computed and verified by the adversary: we deliver the messages without MACs to the adversary, and the adversary can add the MACs; conversely, the adversary can remove the MACs before delivering messages to the protocol model. We let the adversary choose the key $R_r$ that the responder uses for computing cookies. All other elements needed to compute the MACs are public: constants and static public keys. We reintroduce the MACs in a separate model that we use for proving resistance against DoS.

Importantly, these modelling tricks increase the power of the adversary: the implementation done in WireGuard is a particular case of what the adversary can do in our model, in which the adversary chooses the current time as timestamp, increases the counter for sending messages at each emission, accepts incoming counters in a sliding window, and computes and verifies $\mathrm{mac}_1$ and $\mathrm{mac}_2$ by itself. As a result, a security proof in our model remains valid in WireGuard.

## 3.6 VERIFICATION RESULTS

In order to prove authentication properties, we insert events in our model, to indicate when each message is sent or received by the protocol. Specifically, we insert events sent1, sent2, sent_msg_initiator, and sent_msg_responder just before sending message 1, message 2, and transport messages on the

initiator and responder sides respectively, and corresponding events rcvd1, rcvd2, rcvd_msg_responder, and rcvd_msg_initiator when these messages have been received and successfully decrypted. The event rcvd2 and the events for transport messages are executed only in clean sessions.

MUTUAL KEY AND MESSAGE AUTHENTICATION, RESISTANCE AGAINST KCI, RESISTANCE AGAINST REPLAY FROM MESSAGE 2. We show authentication for all messages starting from the second protocol message, by proving the following correspondence properties between events, in the first two variants of our CryptoVerif model of Section 3.5.1:

$$\begin{aligned}&\mathsf{inj\text{-}event}(\mathsf{rcvd2}(S_r^{\mathrm{pub}}, E_i^{\mathrm{pub}}, S_{i\lock}^{\mathrm{pub}}, S_i^{\mathrm{pub}}, \mathrm{ts}_\lock, \mathrm{ts}, E_r^{\mathrm{pub}}, \mathrm{empty}_\lock, T^{\rightarrow}, T^{\leftarrow}))\\ \Rightarrow\ &\mathsf{inj\text{-}event}(\mathsf{sent2}(S_r^{\mathrm{pub}}, E_i^{\mathrm{pub}}, S_{i\lock}^{\mathrm{pub}}, S_i^{\mathrm{pub}}, \mathrm{ts}_\lock, \mathrm{ts}, E_r^{\mathrm{pub}}, \mathrm{empty}_\lock, T^{\rightarrow}, T^{\leftarrow}))\,,\end{aligned}$$

$$\begin{aligned}&\mathsf{inj\text{-}event}(\mathsf{rcvd\_msg\_responder}(S_r^{\mathrm{pub}}, E_i^{\mathrm{pub}}, S_{i\lock}^{\mathrm{pub}}, S_i^{\mathrm{pub}}, \mathrm{ts}_\lock, \mathrm{ts},\\ &\qquad E_r^{\mathrm{pub}}, \mathrm{empty}_\lock, T^{\rightarrow}, T^{\leftarrow}, N^{\rightarrow}, P_\lock, P))\\ \Rightarrow\ &\mathsf{inj\text{-}event}(\mathsf{sent\_msg\_initiator}(S_r^{\mathrm{pub}}, E_i^{\mathrm{pub}}, S_{i\lock}^{\mathrm{pub}}, S_i^{\mathrm{pub}}, \mathrm{ts}_\lock, \mathrm{ts},\\ &\qquad E_r^{\mathrm{pub}}, \mathrm{empty}_\lock, T^{\rightarrow}, T^{\leftarrow}, N^{\rightarrow}, P_\lock, P))\,,\end{aligned}$$

We also prove a third query (similar to the second one above) for transport data messages in the other direction, with events rcvd_msg_initiator and sent_msg_responder. A proven correspondence between two injective events (inj-event) means that each execution of the left-hand event corresponds to a distinct execution of the right-hand event.

The first query means that, if the initiator session is clean and the initiator has received the second message, then the responder sent it, and initiator and responder agree on their static and ephemeral public keys, session keys, timestamp, and communicated ciphertexts. This authenticates the responder to the initiator.

The second and third queries mean that, if the receiver session is clean and the receiver received a transport packet, then a sender sent that transport packet, and the receiver and the sender agree on their static and ephemeral public keys, session keys, timestamp, sent plaintext, message counter, and communicated ciphertexts. In particular, for the key confirmation message, this authenticates the initiator to the responder. These queries also provide message authentication for the transport data messages.

All these properties hold when the pre-shared key is not compromised (variant 2 of Section 3.5.1). They also hold when neither both $S_i^{\mathrm{priv}}$ and $E_i^{\mathrm{priv}}$ nor both $S_r^{\mathrm{priv}}$ and $E_r^{\mathrm{priv}}$ are compromised and the receiver session is clean; this is true, in particular, when the sender's static private key is not compromised yet (variant 1 of Section 3.5.1).

The above queries include resistance against replays because the correspondences are injective: each reception corresponds to a *distinct* emission. They also include resistance against KCI attacks because the rcvd$*$ events are issued even if the receiver's static key has already been compromised: the receiver session is still clean in this case. Note that, for the responder, resistance against KCI attacks only starts after it receives the first data transport message. Indeed, the first protocol message is subject to a KCI attack: if the private static key of the responder ($S_r^{\mathrm{priv}}$) is compromised, then the

adversary can forge the first message and impersonate the initiator to the responder.

SECRECY AND FORWARD SECRECY. We show secrecy of transport data messages in clean sessions by a left-or-right message indistinguishability game. In the initial setup, we randomly choose a secret bit. For each transport data message in a clean session, the adversary provides two padded plaintexts of the same length, and we encrypt one of them depending on the value of that bit. CryptoVerif proves the secrecy of that bit, in variants 1 and 2 of Section 3.5.1, showing that the adversary cannot determine which of the two plaintexts was encrypted.

The secrecy query includes forward secrecy, because we allow dynamic compromise of static keys after the session keys have been established, if the ephemeral key of the same party is not compromised. This assumes that the parties delete the sessions' ephemeral and chaining keys after key derivation.

In variant 2 of our model, the query also shows secrecy provided the pre-shared key is not compromised, even if all other keys (static and ephemeral) are compromised. Our models do not consider the dynamic compromise of the pre-shared key, due to a limitation of CryptoVerif. We can still obtain forward secrecy with respect to the compromise of the pre-shared key using the following manual argument. As mentioned above, variant 2 of our model shows authentication when the pre-shared key is not compromised (all other keys are compromised in this model). This authentication property is preserved when the pre-shared key is compromised after the rcvd$*$ event, because the later compromise cannot alter the fact that the sent$*$ event has been executed. Furthermore, authentication guarantees that the ephemeral public key received by the initiator was generated by the responder and conversely. Variant 1 of our model then guarantees secrecy in this case, because the session is clean when the ephemeral received by the initiator was generated by the responder and conversely. Hence, we get the desired forward secrecy property: we have message secrecy when the pre-shared key is compromised after the session, and neither both $S_i^{\text{priv}}$ and $E_i^{\text{priv}}$ nor both $S_r^{\text{priv}}$ and $E_r^{\text{priv}}$ are compromised.

We cannot prove key secrecy for the session keys in the full protocol, because the session keys are used for encrypting transport data messages, and this allows an adversary to distinguish them from fresh random keys. Instead, we prove key secrecy for a model in which all transport data messages, including key confirmation, are removed. To prove this result, we need to strengthen the session cleanliness condition. Indeed, the first message is subject to a KCI attack, as mentioned above. Therefore, when the private static key of the responder is compromised, we additionally require that the ephemeral received by the responder is equivalent to one generated by the initiator. With this stronger cleanliness condition, we show that the session keys are secret, that is, the keys for various clean sessions are indistinguishable from independent random keys. We do not need this stronger cleanliness condition when we study the full protocol, since the key confirmation message protects the responder against KCI attacks.

RESISTANCE AGAINST REPLAY FOR THE FIRST MESSAGE. We prove that the first message cannot be replayed but only if no static key is compromised when it is received. If $S_i^{\mathrm{priv}}$ were compromised, the adversary can impersonate the initiator as the sender of this message. If $S_r^{\mathrm{priv}}$ is compromised, we have a KCI attack, as described above. So we prove the following injective correspondence in a model where the static keys cannot be compromised but the ephemeral keys may be compromised, so we rely on the static-static Diffie-Hellman shared secret:

$$\begin{aligned}&\mathsf{inj\text{-}event}(\mathsf{rcvd1}(\mathrm{true}, S_r^{\mathrm{pub}}, E_i^{\mathrm{pub}}, S_{i\,\text{🔒}}^{\mathrm{pub}}, S_i^{\mathrm{pub}}, ts_{\text{🔒}}, \mathrm{ts}))\\ \Rightarrow\ &\mathsf{inj\text{-}event}(\mathsf{sent1}(\quad S_r^{\mathrm{pub}}, E_i^{\mathrm{pub}}, S_{i\,\text{🔒}}^{\mathrm{pub}}, S_i^{\mathrm{pub}}, ts_{\text{🔒}}, \mathrm{ts}))\,.\end{aligned}$$

The first parameter of rcvd1 is $\mathrm{true}$ if the public static key received by the responder with the first message is the public static key of the honest initiator: we prove this property only for sessions between honest peers. Replay protection is guaranteed by each timestamp being accepted only once. With this check removed, the first message can be replayed, but we still prove a non-injective correspondence between the two events, replacing inj-event by event in the query. This is a weaker property, meaning that, if an event rcvd1 has been executed, then at least one event sent1 with matching parameters has been executed before. Thus, even with the replay protection removed, we can prove that the origin of the first message cannot be forged in a model without static key compromise.

CORRECTNESS. Correctness means that, if the adversary does not modify the first two messages, then the initiator and responder share the same session keys and transcript hash $H_7$. Actually, it suffices that the adversary does not modify the ephemerals and ciphertexts of the first two messages. We prove it with the following query:

$$\begin{aligned}&\mathsf{event}(\mathsf{responder\_corr}(E_i^{\mathrm{pub}}, S_{i\,\text{🔒}}^{\mathrm{pub}}, \mathrm{ts}_{\text{🔒}}, E_r^{\mathrm{pub}}, \mathrm{empty}_{\text{🔒}}, T_r^{\rightarrow}, T_r^{\leftarrow}, H_{r7}))\\ \wedge\ &\mathsf{event}(\mathsf{initiator\_corr}(E_i^{\mathrm{pub}}, S_{i\,\text{🔒}}^{\mathrm{pub}}, \mathrm{ts}_{\text{🔒}}, E_r^{\mathrm{pub}}, \mathrm{empty}_{\text{🔒}}, T_i^{\rightarrow}, T_i^{\leftarrow}, H_{i7}))\\ \Rightarrow\ &T_i^{\rightarrow} = T_r^{\rightarrow} \wedge T_i^{\leftarrow} = T_r^{\leftarrow} \wedge H_{i7} = H_{r7}\,.\end{aligned}$$

The events initiator_$*$ and responder_$*$ used in this query and in the following ones are issued after key derivation, in the initiator and responder respectively. Here, the two events given as assumptions guarantee that the adversary did not modify the ephemerals and ciphertexts of the first two messages, and the query concludes that the session keys and transcript hash must be equal. However, in our main models, CryptoVerif is currently unable to prove that the ciphertexts have not been created by the adversary, although this is true in the sessions considered by the correctness query. Thus, we created a separate model to prove correctness, in which the assumption is hard-coded by interleaving the initiator and responder in a single sequential process. In this model, we prove correctness even if all keys are compromised.

SESSION UNIQUENESS. First, we prove that there is a single initiator and a single responder session with a given $T^{\rightarrow}$ or $T^{\leftarrow}$. The query below shows

that there cannot be two distinct initiator sessions with the same $T^{\rightarrow}$:

$$\begin{aligned}&\mathsf{event}(\mathsf{initiator\_uniq\_T}^{\rightarrow}(i_i, T^{\rightarrow}))\\ \wedge\ &\mathsf{event}(\mathsf{initiator\_uniq\_T}^{\rightarrow}(i_i', T^{\rightarrow})) \Rightarrow i_i = i_i',\end{aligned}$$

where $i_i, i_i'$ are replication indices: CryptoVerif assigns each execution of the initiator (or responder) process a unique replication index, so the query means that if we execute two events $\mathsf{initiator\_uniq\_T}^{\rightarrow}$ with the same $T^{\rightarrow}$, then they have the same replication index $i_i = i_i'$, hence they belong to the same session. This query is proved in variant 3 of Section 3.5.1, so the property holds even if all keys are compromised. (It relies on the choice of a fresh ephemeral at each session.) The queries for the other cases are similar.

Second, we show similarly that there is a single initiator and a single responder session for a given set of publicly transmitted protocol values.

CHANNEL BINDING. We prove channel binding with the query:

$$\begin{aligned}&\mathsf{event}(\mathsf{initiator\_H7}(\mathrm{params}, H_7))\\ \wedge\ &\mathsf{event}(\mathsf{responder\_H7}(\mathrm{params}', H_7)) \Rightarrow \mathrm{params} = \mathrm{params}'\end{aligned}$$

This query shows that if the initiator and responder have the same value of the session transcript $H_7$, then they share the same value of all session parameters $\mathrm{params}$ (static and ephemeral public keys, timestamp, pre-shared key, session keys). This query is also proved in variant 3 of Section 3.5.1, so the property holds even if all keys are compromised. (It relies on the collision resistance of hash.)

IDENTITY MIS-BINDING. For this property, we need to show that if an initiator and a responder session share the same session keys $T^{\rightarrow}$ and $T^{\leftarrow}$, then they share the same view on the ephemeral and static keys used in that session. This is formalised by the following query:

$$\begin{aligned}&\mathsf{event}(\mathsf{responder\_imb}(T^{\rightarrow}, T^{\leftarrow}, E_{i,\mathrm{rcvd}}^{\mathrm{pub}}, E_r^{\mathrm{pub}}, S_{i,\mathrm{rcvd}}^{\mathrm{pub}}, S_r^{\mathrm{pub}}))\\ \wedge\ &\mathsf{event}(\mathsf{initiator\_imb}(T^{\rightarrow}, T^{\leftarrow}, E_i^{\mathrm{pub}}, E_{r,\mathrm{rcvd}}^{\mathrm{pub}}, S_i^{\mathrm{pub}}, S_{r,\mathrm{rcvd}}^{\mathrm{pub}}))\\ \Rightarrow\ &E_i^{\mathrm{pub}} = E_{i,\mathrm{rcvd}}^{\mathrm{pub}} \wedge E_r^{\mathrm{pub}} = E_{r,\mathrm{rcvd}}^{\mathrm{pub}} \wedge S_i^{\mathrm{pub}} = S_{i,\mathrm{rcvd}}^{\mathrm{pub}} \wedge S_r^{\mathrm{pub}} = S_{r,\mathrm{rcvd}}^{\mathrm{pub}}.\end{aligned}$$

CryptoVerif proves it in variant 1 of our model, so it holds when neither both $S_i^{\mathrm{priv}}$ and $E_i^{\mathrm{priv}}$ nor both $S_r^{\mathrm{priv}}$ and $E_r^{\mathrm{priv}}$ are compromised. However, the proof fails when all static and ephemeral keys are compromised (variant 3 of our model): CryptoVerif can prove only the weaker property that $\mathrm{pow\_k}\left(S_i^{\mathrm{pub}}\right) = \mathrm{pow\_k}\left(S_{i,\mathrm{rcvd}}^{\mathrm{pub}}\right)$ and $\mathrm{pow\_k}\left(S_r^{\mathrm{pub}}\right) = \mathrm{pow\_k}\left(S_{r,\mathrm{rcvd}}^{\mathrm{pub}}\right)$. An adversary can indeed break the equality of public static keys in this case:

- The adversary instructs $A$ to initiate a session to a public static key $S_r^{\mathrm{pub}\prime}$ equivalent to our model's honest responder public static key: $\mathrm{pow\_k}(S_r^{\mathrm{pub}}) = \mathrm{pow\_k}(S_r^{\mathrm{pub}\prime})$ but $S_r^{\mathrm{pub}} \neq S_r^{\mathrm{pub}\prime}$. This is possible because $S_r^{\mathrm{priv}}$ is compromised. In this session, the adversary acts as responder, and because the ephemeral is also compromised, gets $A$'s $E_i^{\mathrm{priv}}$.
- The adversary now acts as initiator to start a session with $B$ using a public static key $S_i^{\mathrm{pub}\prime}$ equivalent to the honest initiator public static

key: $\mathrm{pow\_k}(S_i^{\mathrm{pub}}) = \mathrm{pow\_k}(S_i^{\mathrm{pub}\prime})$ but $S_i^{\mathrm{pub}} \neq S_i^{\mathrm{pub}\prime}$. This is possible because $S_i^{\mathrm{priv}}$ is compromised. The adversary uses $E_i^{\mathrm{priv}}$ as ephemeral. The ephemeral of this session is also compromised, so the adversary gets $E_r^{\mathrm{priv}}$.

- The adversary continues the session with $A$ using the ephemeral $E_r^{\mathrm{priv}}$.

If a pre-shared key is used, we assume that the adversary has the same pre-shared key with $A$ (presenting itself with key $S_r^{\mathrm{pub}\prime}$) and with $B$ (presenting itself with $S_i^{\mathrm{pub}\prime}$). The session keys $T^{\rightarrow}$ and $T^{\leftarrow}$ for these two sessions are computed as hashes of $E_i^{\mathrm{pub}}$, $\mathsf{dh}(E_i^{\mathrm{priv}}, S_r^{\mathrm{pub}})$, $\mathsf{dh}(S_i^{\mathrm{priv}}, S_r^{\mathrm{pub}})$, $E_r^{\mathrm{pub}}$, $\mathsf{dh}(E_i^{\mathrm{priv}}, E_r^{\mathrm{pub}})$, $\mathsf{dh}(S_i^{\mathrm{priv}}, E_r^{\mathrm{pub}})$, and $\mathrm{psk}$. They are the same in both sessions, so the session keys are also the same.

This scenario, with a session between $A$ and $B'$ and one between $B$ and $A'$ that share the same session keys, is an instance of a bilateral unknown key-share attack [CT08] and of a key synchronisation attack [Bha+14a]. It appears only when all static and ephemeral Diffie-Hellman keys are compromised, and hence should be considered a corner-case. However, we note that this scenario does not require the $\mathrm{psk}$ shared by $A$ and $B$ to be compromised, since this $\mathrm{psk}$ does not get used in the execution above. We suggest a possible fix of this identity mis-binding issue in Section 3.7.

[CT08] Chen and Tang, "Bilateral Unknown Key-Share Attacks in Key Agreement Protocols"

[Bha+14a] Bhargavan et al., "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS"

RESISTANCE AGAINST DOS. As described in Section 3.2, WireGuard provides a cookie mechanism that a peer under load can use to enforce a round trip per sender address, and thus to bind a handshake message to a sender address; this permits per-address rate limiting. We model this mechanism in a separate model in which a responder generates $R_r$, replies with a cookie $\tau = \mathsf{mac}(R_r, A_i)$ upon receipt of messages 1 from $A_i$ with zero $\mathrm{mac}_2$, and verifies $\mathrm{mac}_2$ upon receipt of messages 1 with non-zero $\mathrm{mac}_2$. The rest of the protocol is run by the adversary, which has the long-term static keys. In particular, we do not model the encryption of the cookie $\tau$, but send it in the clear, assuming that the adversary carries out the encryption and decryption, which depend only on values it knows.

In this model, we prove that, if a responder under load accepts a handshake message from a sender with address $A_i$, then this sender passed through a round trip, that is, the responder did indeed previously generate a cookie for the address $A_i$. This formalised by the following query:

$$\begin{aligned} &\mathsf{event}(\mathsf{accepted\_cookie}(A_i, i_r, \tau, \mathrm{msg}_\beta, \mathrm{mac}_2)) \\ \Rightarrow\ &\mathsf{event}(\mathsf{generated\_cookie}(A_i, i_r, \tau))\,, \end{aligned}$$

where $i_r$ is an index that uniquely identifies the key $R_r$ used for generating the cookie. This query is proved under the assumption that mac is a pseudo-random function (PRF).

IDENTITY HIDING. When the adversary has a candidate public key $S_Y^{\mathrm{pub}}$, it can determine whether this public key is involved in WireGuard sessions, as already mentioned in the WireGuard specification [Don17]. In the first message, it can test whether $\mathrm{mac}_1 = \mathsf{mac}(\mathsf{hash}(\mathsf{label}_{\mathsf{mac1}} \| S_Y^{\mathrm{pub}}), \mathrm{msg}_\alpha)$ and that reveals whether $S_Y^{\mathrm{pub}} = S_r^{\mathrm{pub}}$. A similar test on message 2 reveals whether $S_Y^{\mathrm{pub}} = S_i^{\mathrm{pub}}$. When an entity with public key $S_m^{\mathrm{pub}}$ sends a cookie reply, the adversary can try to decrypt the encrypted cookie $\tau_{\text{🔒}}$ with the key

[Don17] Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel"

$\mathsf{hash}(\mathsf{label}_{\mathsf{cookie}}\|S_Y^{\mathrm{pub}})$, the nonce $\mathrm{nonce}$ (obtained from the cookie reply), and the associated data $\mathrm{mac}_1$ (obtained from a previous message). If decryption succeeds, then the adversary knows that $S_Y^{\mathrm{pub}} = S_m^{\mathrm{pub}}$. In practice, the public keys of VPN endpoints may be easy to obtain: they are often published to subscribers on a web page. In such scenarios, WireGuard does not provide identity hiding.

If we consider the protocol without MACs and cookie reply, that is, basically the Noise protocol IKpsk2, we can obtain stronger identity protection guarantees, however with the additional assumption that the AEAD scheme also preserves the secrecy of the associated data. Indeed, if the AEAD scheme is only IND-CPA and INT-CTXT, then the adversary may obtain the associated data of the first ciphertext $S_i^{\mathrm{pub}}$, that is, $\mathsf{hash}(\mathsf{hash}(H_0\|S_r^{\mathrm{pub}})\|E_i^{\mathrm{pub}})$. It can compare this value with $\mathsf{hash}(\mathsf{hash}(H_0\|S_Y^{\mathrm{pub}})\|E_i^{\mathrm{pub}})$ since $E_i^{\mathrm{pub}}$ is sent in the first message and $H_0$ is a constant. Thus, it can determine whether $S_r^{\mathrm{pub}} = S_Y^{\mathrm{pub}}$.

However, assuming that the AEAD scheme also preserves the secrecy of the associated data, we prove using CryptoVerif that the protocol without MACs and cookie reply satisfies the following identity hiding property: an adversary that has $S_{A1}^{\mathrm{pub}}, S_{A2}^{\mathrm{pub}}, S_{B1}^{\mathrm{pub}}, S_{B2}^{\mathrm{pub}}$ cannot distinguish a configuration in which the entity with public key $S_{A1}^{\mathrm{pub}}$ initiates sessions with $S_{B1}^{\mathrm{pub}}$ from one in which the entity with public key $S_{A2}^{\mathrm{pub}}$ initiates sessions with $S_{B2}^{\mathrm{pub}}$. ChaCha20Poly1305 indeed preserves the secrecy of the associated data, because it satisfies the stronger IND$-CPA property, which requires the ciphertext to be indistinguishable from random bits, as shown in [Pro14].

[Pro14] Procter, *A Security Analysis of the Composition of ChaCha20 and Poly1305*

We discuss possible solutions to strengthen the identity hiding for the protocol with MACs in Section 3.7.

PROOF GUIDANCE AND METRICS. CryptoVerif needs to be manually guided to perform these proofs. We detail the instructions given to CryptoVerif for proving authentication and message secrecy in variant 1 of our model, with dynamic compromise of the private static keys. The guidance we give for other proofs follows similar ideas.

First, we set some options, in particular to speed up the proof and save memory.

```
set casesInCorresp = false;
```

reduces the number of cases that CryptoVerif considers in proofs of correspondences. This option does not affect the soundness; in complex cases, CryptoVerif might just not be able to prove a correspondence with this option set to false.

```
set mergeBranches = false;
```

prevents CryptoVerif from trying to automatically merge branches of tests when they execute the same code.

```
set forgetOldGames = true;
```

tells CryptoVerif to remove games generated by previous instructions from memory, in order to save memory. (However, that prevents undoing previous proof steps.)

```
set useKnownEqualitiesWithFunctionsInMatching = true;
```

tells CryptoVerif to apply known equalities that start with a function symbol when it tests whether a term matches another term. CryptoVerif would not do that by default because it is costly. However, when using Curve25519, we need to apply equalities of the form $\mathrm{pow\_k}(\cdot) = \mathrm{pow\_k}(\cdot)$, so we use this setting. We unset it later when it is no longer needed, to speed up the proof.

Next, we distinguish cases. In the initiator *A*, we add a test to distinguish whether the partner public key `S_X_pub` is equivalent to *B*'s static public key `S_B_pub`. This test is added just after the input that receives `S_X_pub` from the adversary on channel `c_config_initiator`, by the instruction:

```
insert after "in(c_config_initiator\\["
  "if pow_k(S_X_pub) = pow_k(S_B_pub) then";
```

Here, the test `if pow_k(S_X_pub) = pow_k(S_B_pub) then` is inserted after the line that contains the regular expression `in(c_config_initiator\\[`. (the character sequence `\\[` denotes the character `[` in regular expressions.) This is an improvement that we implemented in CryptoVerif. Before, CryptoVerif required indicating the program point at which a case distinction should be inserted by an integer number, and this number often varied with very minor changes in the protocol specification. We modified CryptoVerif to allow specifying program points as the beginning of a line that matches a certain regular expression, or as the line that follows a matching line. This is much more stable to small changes in the protocol model.

As a result, the initiator ephemeral is then chosen at two different places, in the `then` branch and in the `else` branch of the test that was just introduced. We rename the variable `E_i_pub_4` containing the initiator public ephemeral to two distinct names (these names are chosen by CryptoVerif and are here `E_i_pub_6` and `E_i_pub_7`) by the instruction:

```
SArename E_i_pub_4;
```

In the responder *B*, we distinguish whether the partner public key `S_i_pub_rcvd_4` is equivalent to *A*'s static public key `S_A_pub`, by the instruction:

```
insert after
  "let injbot(G_to_bitstring(S_i_pub_rcvd_4: G_t))"
  "if pow_k(S_i_pub_rcvd_4) = pow_k(S_A_pub) then";
```

The test is inserted after the decryption of the ciphertext $S_{i\,\text{🔒}}^{\mathrm{pub}}$. In the responder *B*, we also distinguish whether the received ephemeral `E_i_pub_rcvd_3` is equivalent to an ephemeral generated by *A*, `E_i_pub_6[i]` or `E_i_pub_7[i]` for any `i`. (`E_i_pub_6` and `E_i_pub_7` are arrays containing one public ephemeral for each execution of the *A*.) This test is inserted after the reception of the ephemeral by the responder *B*, by the following instruction:

```
insert after "in(c_init2resp_recv\\["
  "find i <= N_init_parties suchthat defined(E_i_pub_7[i])
     && pow_k(E_i_pub_rcvd_3) = pow_k(E_i_pub_7[i]) then
 orfind i <= N_init_parties suchthat defined(E_i_pub_6[i])
     && pow_k(E_i_pub_rcvd_3) = pow_k(E_i_pub_6[i]) then";
```

The construct `find` $i$ `<=` $N$ `suchthat defined(`$x[i]$`) &&` $M$ `then` $P$ `else` $P'$ looks for an index $i$ such that $x[i]$ is defined and the condition $M$ holds. If it finds one, it runs $P$ with that index; otherwise, it runs $P'$. It is extended to several branches by using `orfind`. Finally, in the initiator $A$, we distinguish whether the responder's ephemeral `E_r_pub_rcvd_2` received with the second protocol message is equivalent to an ephemeral generated by $B$, `E_r_pub[j]`, by the instructions:

```
insert after_nth 2 "in(c_resp2init_recv\\["
  "find j <= N_resp_parties suchthat defined(E_r_pub[j]) &&
     pow_k(E_r_pub_rcvd_2) = pow_k(E_r_pub[j]) then";
insert after_nth 1 "in(c_resp2init_recv\\["
  "find j <= N_resp_parties suchthat defined(E_r_pub[j]) &&
     pow_k(E_r_pub_rcvd_2) = pow_k(E_r_pub[j]) then";
```

We insert two tests because we need to insert one test in each branch of the initial case distinction made in the initiator. These case distinctions allow us to isolate Diffie-Hellman shared secrets that the adversary will be unable to compute because both shares come from honest participants. We simplify the obtained game by

```
simplify;
```

Then, we apply the random oracle assumption for the 3 random oracles $\mathsf{chain}'_6$, $\mathsf{chain}'_2$, $\mathsf{chain}'_1$ (named `rom3_intermediate`, `rom2_intermediate`, and `rom1_intermediate` in the CryptoVerif file). For the arguments of these oracles that are Diffie-Hellman shared secrets in the protocol (and thus are in $G_{\mathrm{sub}}$), we distinguish whether the argument received by the random oracle from the adversary is in $G_{\mathrm{sub}}$ before applying the random oracle assumption. (When it is not in $G_{\mathrm{sub}}$, it cannot collide with a call coming from the protocol.) This is done by the following instructions:

```
insert after "in(ch1_rom3"
  "let rom3_input(x1_rom3, Gsub_to_G(x2_rom3),
     Gsub_to_G(x3_rom3), x4_rom3, Gsub_to_G(x5_rom3),
     Gsub_to_G(x6_rom3), v_psk) = x_rom3 in";
crypto rom(rom3_intermediate);
insert after "in(ch1_rom2"
  "let rom2_input(x1_rom2, Gsub_to_G(x2_rom2),
     Gsub_to_G(x3_rom2)) = x_rom2 in";
crypto rom(rom2_intermediate);
insert after "in(ch1_rom1"
  "let rom1_input(x1_rom1, Gsub_to_G(x2_rom1)) = x_rom1 in";
crypto rom(rom1_intermediate);
```

The first case distinction distinguishes whether the argument `x_rom3` of `rom3_intermediate` is of the form `rom3_input(x1_rom3, Gsub_to_G(x2_rom3), Gsub_to_G(x3_rom3), x4_rom3, Gsub_to_G(x5_rom3), Gsub_to_G(x6_rom3), v_psk)`, that is, a tuple in which the 2nd, 3rd, 5th, and 6th components are in $G_{\mathrm{sub}}$. The next instruction applies the random oracle assumption to `rom3_intermediate`. The other two random oracles are handled similarly.

Next, we apply the gap Diffie-Hellman assumption to exp_div_k; the associated private keys are the private static and ephemeral keys of the initiator and the responder:

```
crypto gdh(exp_div_k)
  S_B_priv E_i_priv_8 S_A_priv E_r_priv_4;
```

We modify settings to speed up the rest of the proof by the following instructions:

```
set useKnownEqualitiesWithFunctionsInMatching = false;
set elsefindFactsInSimplify = false;
```

The setting `elsefindFactsInSimplify`, when true, tells CryptoVerif to simplify games using the information obtained from being in an `else` branch of a `find`. It is the default, but it can be costly for large games.

We split the keys generated by $\mathsf{chain}'_6$ into 4 keys by

```
crypto splitter(concat_four_keys) **;
```

The indication $**$ means that we apply `splitter(concat_four_keys)` as many times as we can, without performing a full simplification between each application. Avoiding that simplification speeds up the proof a bit. `splitter(concat_four_keys)` means that a random bitstring of length 4 times the length of a key is indistinguishable from the concatenation of 4 random keys.

By default, when a cryptographic transformation fails, CryptoVerif tries to determine syntactic transformations that might make it succeed, applies those transformations, and retries the cryptographic transformation. For speed, we disable this behaviour by the following instruction:

```
set noAdviceCrypto = true;
```

We apply ciphertext integrity of the AEAD scheme:

```
crypto int_ctxt(enc) *;
```

The indication $*$ means that we apply `int_ctxt(enc)` as many times as we can. Then we try to prove security properties:

```
success;
```

CryptoVerif shows the impossibility of nonce reuse in the AEAD scheme and the absence of identity mis-binding attacks. We simplify the game

```
simplify;
```

For keys that the adversary may have after compromising the static keys, we apply a variant of the ciphertext integrity transformation that allows corruption, as follows:

```
crypto int_ctxt_corrupt(enc) k_51;
```

The key k_51 is generated by the initiator when the partner static key is equivalent to $B$'s static public key but the received ephemeral is not equivalent to an ephemeral generated by $B$, and $B$'s static key is not compromised. In this case, the adversary cannot produce a valid ciphertext in protocol message 2 (empty plaintext), thus the decryption will fail on the initiator's side and the protocol will not continue.

Then we try to prove security properties:

```
success;
```

CryptoVerif proves that the initiator can authenticate the second protocol message as well as transport data messages sent by the responder. We simplify the game

```
simplify;
```

and again apply the variant of the ciphertext integrity transformation that allows corruption:

```
crypto int_ctxt_corrupt(enc) "T_i_send_[0-9]*";
```

We apply this transformation to all keys of the form T_i_send_$n$ for integers $n$. We want to apply this transformation to keys generated by the responder, in case the partner public key is equivalent to $A$'s static public key, but the received ephemeral is not equivalent to an ephemeral generated by $A$, and $A$'s static key is not compromised. In this case, the adversary cannot produce a valid ciphertext for a transport data message, thus the decryption will fail on the responder's side and the protocol will not continue. The keys in question are many variables of the form T_i_send_$n$; we apply the transformation to all variables of this form as it is easier and the proof still works. Then we try to prove security properties:

```
success;
```

CryptoVerif shows that the responder can authenticate transport data messages sent by the initiator. We again simplify the game

```
simplify;
```

That removes all events, which are no longer useful since all correspondence properties are proved. Then we apply the IND-CPA property of the AEAD scheme as many times as we can:

```
crypto ind_cpa(enc) **;
```

and finally prove message secrecy:

```
success
```

In total, we give 36 instructions to CryptoVerif to perform this proof (not counting the instruction to display the current game), and CryptoVerif generates a sequence of 168 games. This proof takes 17 min, the proof of key secrecy with dynamic compromise of private static keys takes 19 min, and the one for identity hiding 26 min on one core of an Intel Xeon 3.6 GHz; these are our longest proofs.

## 3.7 DISCUSSION

WireGuard is a promising new VPN protocol that aims to replace IPsec and OpenVPN, and is being considered for adoption within the Linux kernel. We presented a mechanised cryptographic proof for a detailed model of WireGuard using the CryptoVerif prover. Our model accounts for the full Noise IKpsk2 secure channel protocol as well as WireGuard's extensions for stealthy operation and DoS resistance. We consider an arbitrary number of parallel sessions, with an arbitrary number of transport data messages. Furthermore, we base our proof on a precise model of the Curve25519 group.

We proved correctness, message and key secrecy, forward secrecy, mutual authentication, session uniqueness, channel binding, and resistance against replay, key compromise impersonation, and denial of service attacks. In some cases, our analysis pointed out potential improvements in the protocol (which we did not prove secure using CryptoVerif):

ADDING PUBLIC KEYS TO THE CHAINING KEY DERIVATION. When analysing WireGuard for Identity Mis-Binding attacks, our analysis uncovered a corner case. Suppose all the Diffie-Hellman keys in a session between two hosts $A$ and $B$ were compromised, but the pre-shared key between them is still secret. Then the adversary can set up a man-in-the-middle attack where $A$ thinks it is connected to $B'$, $B$ thinks it is connected to $A'$, but in fact they are both connected to each other, in the sense that the two connections have the same traffic keys, even though they have different static keys.

In particular, once it has set up the session, the adversary can step away and let $A$ and $B$ directly communicate with each other, while retaining the ability to read and modify messages at will. Interestingly, this vulnerability only appears in our precise model of Curve25519; it cannot be detected under a classic Diffie-Hellman assumption.

Although this attack scenario may be quite unrealistic, it points to a theoretical weakness in the protocol that is easy to prevent with a simple modification. Noise IKpsk2 already adds ephemeral public keys to the chaining key derivation; we recommend that the static public keys be added as well. Alternatively, adding the full transcript hash to the traffic key derivation would also prevent this corner case.

Separately, it is also worth noting that adding public keys to the key derivation significantly helps with the cryptographic proof. For example, consider the Noise IK protocol, which is similar to IKpsk2 except that it does not use PSKs. IK does not mix the ephemeral keys into the chaining key, and it turns out that it is much harder for CryptoVerif to verify than IKpsk2, since we now have to reason about mis-matched ephemeral keys. In particular, even if we use a public PSK key of all-zeroes, the IKpsk2 protocol is easier to prove secure than IK. In fact, our recommendation is to add further contextual information to the key derivation. It would not only prevent theoretical attacks, but also make proofs easier.

BALANCING STEALTH AND IDENTITY HIDING. Our analysis also points out that the use of static public keys in $\mathrm{mac}_1$ and $\mathrm{mac}_2$ in WireGuard negatively

affects the identity hiding guarantees provided by IKpsk2. This is a conscious trade-off that WireGuard makes to achieve stealthy operation [Don17]. However, in deployment scenarios where identity hiding is more important than stealth, we recommend that the protocol use a constant (say all-zeroes) instead of the static public keys to compute the MACs and cookies.

While it is difficult to preserve stealth while hiding the responder's identity, a modification to the protocol can still hide the initiator's identity. We recommend that the initiator should send a MAC key (along with the timestamp) in the first handshake message, and the responder should use this MAC key to compute $\mathrm{mac}_1$ in the second handshake message. The initiator can verify this MAC to get DoS protection, but its static public key is kept hidden from a network adversary. Essentially, the MAC key acts as an in-session cookie.

[Don17] Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel"

RELATED WORK. The use of formal verification tools to analyse real-world cryptographic protocols is now a well-established research area with hundreds of case studies (see e.g. [Bla12]). CryptoVerif itself has been used to analyse modern protocols like Signal [KBB17] and TLS 1.3 [BBK17]. We conclude this chapter by comparing our results with closely related work; Table 3.1 provides a condensed, high-level overview.

[Bla12] Blanchet, "Security Protocol Verification: Symbolic and Computational Models"

[KBB17] Kobeissi et al., "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach"

[BBK17] Bhargavan et al., "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate"

WireGuard itself has been formally analysed before. Donenfeld et al. symbolically analyse the IKpsk2 key exchange protocol used by WireGuard for a number of security goals, including identity mis-binding and identity hiding [DM18]. However, they do not model the MACs or the cookie mechanism, and hence they do not prove DoS resistance. Interestingly, their analysis concludes the absence of identity mis-binding attacks even if all keys are compromised, because their model does not include equivalent public keys. We disprove this property by considering a precise model of Curve25519. An improved modelling of Diffie-Hellman groups in the symbolic model has independently been proposed, using Tamarin [CJ19]. It could probably be used to improve the symbolic analysis of WireGuard.

[DM18] Donenfeld and Milner, *Formal Verification of the WireGuard Protocol*

[CJ19] Cremers and Jackson, "Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman"

Dowling and Paterson [DP18] present a manual cryptographic analysis of WireGuard. In particular, they prove key indistinguishability for the WireGuard handshake based on the PRF-ODH assumption in an extension of the eCK-PFS key exchange model. (Because of this difference in the used assumption, our mechanisation cannot be used directly to find issues in proof steps; it is a different proof.) Key indistinguishability no longer holds once the key is used, so they prove security for a slightly modified variant of the IKpsk2 protocol that includes a key confirmation message independent of the session keys. In contrast, our proof requires no changes to the protocol, since we use an ACCE-style model. Furthermore, [DP18] focuses only on the key exchange, and does not consider other properties like identity hiding or DoS resistance. Their analysis also does not find the identity mis-binding issue since they do not consider a scenario where all Diffie-Hellman keys are compromised.

[DP18] Dowling and Paterson, "A Cryptographic Analysis of the WireGuard Protocol"

Finally, the Noise Explorer tool [KNB19] has been used to perform a comprehensive symbolic analysis of numerous Noise protocols using the ProVerif analyser. Noise Explorer can be used to find violations of secrecy and authentication properties for any protocol expressed in the language defined

[KNB19] Kobeissi et al., "Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols"

TABLE 3.1: Security models (upper part) and properties analysed (lower part) in different works on WireGuard or Noise IKpsk2.

| | Noise Explorer [KNB19] | Suter-Dörig [SD18] | Girol [Gir19] | Donenfeld, Milner [DM18] | Dowling, Paterson [DP18] | this work |
|---|---|---|---|---|---|---|
| verified protocol | Noise IKpsk2 | | | WireGuard | | |
| tool set | PV | T | T | T | m | CV |
| computational model | x | x | x | x | ✓ | ✓ |
| Curve25519 with equivalent keys | x | x | x | x | x | ✓ |
| compromise static keys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| compromise ephemeral keys | x | x | ✓ | ✓ | ✓ | ✓ |
| dishonest ephemeral keys | x | x | ✓ | x | x | x |
| compromise pre-shared key | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| compromise all keys | x | x | ✓ | ✓ | x | ✓ |
| both roles per static key | x | ✓ | ✓ | ✓ | ✓ | ✓ |
| mutual authentication | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| key compromise impersonation | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 1st message replay | — | — | — | x | x | ✓ |
| transport data replay | x | ✓ | ✓ | x | x | ✓ |
| session uniqueness | x | ✓ | x | ✓ | ✓ | ✓ |
| channel binding | x | ✓ | x | x | x | ✓ |
| DoS resistance | — | — | — | x | x | ✓ |
| forward key secrecy | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| forward message secrecy | ✓ | ✓ | ✓ | x | x | ✓ |
| identity hiding | x | x | ✓ | ✓[2] | x | ✓ |
| identity mis-binding | x | x | x | ✓[1] | x | ✓ |

Definitions differ between models.
T = Tamarin, PV = ProVerif, CV = CryptoVerif, m = manual.
✓= included, x = not included, — = not applicable.
1) The identity mis-binding issue we found was *not* found.
2) Weaker identity hiding property using a surrogate term.

by Noise, using per-message authentication and confidentiality grades. It includes a symbolic analysis of Noise IKpsk2. A similar work has been done in Tamarin [SD18; Gir19].

[SD18] Suter-Dörig, "Formalizing and verifying the security protocols from the Noise framework"

[Gir19] Girol, "Formalizing and Verifying the Security Protocols from the Noise Framework"

ACKNOWLEDGEMENTS

We thank Jason A. Donenfeld (the author of WireGuard), Nadim Kobeissi, and the anonymous reviewers of EuroS&P'19 for their helpful feedback on our work. This research was partly funded by the European Unions Horizon 2020 NEXTLEAP Project (grant agreement no. 688722), ERC CIRCUS (grant agreement no. 683032), ANR AnaStaSec (decision number ANR-14-CE28-0014-01), and ANR TECAP (decision number ANR-17-CE39-0004-03).

# 4

# *Analysing HPKE's Authenticated Mode*

This chapter and the appendices belonging to it are based on the long version [Alw+20] of the paper "Analysing the HPKE Standard", published at IACR's Eurocrypt 2021 with co-authors Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, and Doreen Riepel [Alw+21].

[Alw+21] Alwen et al., "Analysing the HPKE Standard"

ABSTRACT. The *Hybrid Public Key Encryption* (HPKE) scheme is a standard developed by the Crypto Forum Research Group (CFRG) of the Internet Research Task Force (IRTF), and has been published as RFC 9180 in February 2022 [Bar+22].

In this chapter, of the four modes of HPKE, we analyse the authenticated mode $\mathsf{HPKE}_{\mathsf{Auth}}$ in its single-shot encryption form as it contains what is, arguably, the most novel part of HPKE. An analysis of all modes including $\mathsf{HPKE}_{\mathsf{Auth}}$ is described in Section 5.1.

$\mathsf{HPKE}_{\mathsf{Auth}}$'s intended application domain is captured by a new primitive which we call Authenticated Public Key Encryption (APKE). We provide syntax and security definitions for APKE schemes, as well as for the related Authenticated Key Encapsulation Mechanisms (AKEMs). We prove security of the AKEM scheme DH-AKEM underlying $\mathsf{HPKE}_{\mathsf{Auth}}$ based on the Gap Diffie-Hellman assumption and provide general AKEM/DEM composition theorems with which to argue about $\mathsf{HPKE}_{\mathsf{Auth}}$'s security. To this end, we also formally analyse $\mathsf{HPKE}_{\mathsf{Auth}}$'s key schedule and key derivation functions. To increase confidence in our results we use the automatic theorem proving tool CryptoVerif. All our bounds are quantitative and we discuss their practical implications for $\mathsf{HPKE}_{\mathsf{Auth}}$.

As an independent contribution we propose the new framework of *nominal groups* that allows us to capture abstract syntactical and security properties of practical elliptic curves, including the Curve25519 and Curve448 based groups (which do not constitute cyclic groups).

[Bar+22] Barnes et al., *Hybrid Public Key Encryption*

## 4.1 INTRODUCTION

At the time of writing of the paper, an effort was underway by the Crypto Forum Research Group (CFRG) to agree upon a new open standard for public key encryption [Bar+22]. The standard is called *Hybrid Public Key Encryption* (HPKE) and it is, in particular, expected to be used as a building block by the Internet Engineering Task Force (IETF) in at least two further

upcoming standardized security protocols [Bar+20a; Res+20]. The primary source for HPKE is an RFC [Bar+22] (on draft 8 [Bar+20b] at the time this analysis was done) which lays out the details of the construction and provides some rough intuition for its security properties.

At first glance the HPKE standard might be thought of as a "public key encryption" scheme in the spirit of the KEM/DEM paradigm [CS03]. That is, it combines a Key Encapsulation Mechanism (KEM) and an Authenticated Encryption with Associated Data (AEAD) acting as a Data Encapsulation Mechanism (DEM) according to the KEM/DEM paradigm. However, upon closer inspection HPKE turns out to be more complex than this perfunctory description implies.

First, HPKE actually consists of 2 different KEM/DEM constructions. Moreover, each construction can be instantiated with a pre-shared key (PSK) known to both sender and receiver, which is used in the key schedule to derive the DEM key. In total this gives rise to 4 different *modes* for HPKE. The *basic* mode $\mathsf{HPKE}_{\mathsf{Base}}$ makes use of a standard (say IND-CCA-secure) KEM to obtain a "message privacy and integrity" only mode. This mode can be extended to $\mathsf{HPKE}_{\mathsf{PSK}}$ to support authentication of the sender via a PSK.

The remaining 2 HPKE modes make use of a different KEM/DEM construction built from a rather non-standard KEM variant which we call an *Authenticated KEM* (AKEM). Roughly speaking, an AKEM can be thought of the KEM analogue of signcryption [Zhe97]. In particular, sender and receiver both have their own public/private keys. Each party requires their own private and the other party's public key to perform en/decryption. The HPKE RFC constructs an AKEM based on a generic Diffie-Hellman group. It goes on to fix concrete instantiations of such groups using either the P-256, P-384, or P-521 NIST curves [Nat13] or the Curve25519 or Curve448 curves [LHT16]. The AKEM-based HPKE modes also intend to authenticate the sender to the receiver. Just as in the KEM-based case, the AKEM/DEM construction can be instantiated in modes either with or without a PSK. We refer to the AKEM/DEM-based mode without a PSK as the *authenticated mode* and, for reasons described below, it is the main focus of this work. The corresponding HPKE scheme is called $\mathsf{HPKE}_{\mathsf{Auth}}$.

Orthogonal to the choice of mode in use, HPKE also provides a so called single-shot and a multi-shot API. The single-shot API can be thought of as pairing a single instance of the DEM with a KEM ciphertext while the multi-shot API establishes a key schedule allowing a single KEM shared secret to be used to derive keys for an entire sequence of DEMs. Finally, HPKE also supports exporting keys from the key schedule for use by arbitrary higher-level applications.

APPLICATIONS. As an open standard of the IRTF, we believe HPKE to be an interesting topic of study in its own right. Indeed, HPKE is already slated for use in at least two upcoming protocols; the Messaging Layer Security (MLS) [Bar+20a] secure group messaging protocol and the Encrypted Server Name Indication (ESNI) extension for TLS 1.3 [Res+20]. Both look to be well-served by the single-shot API as they require a single DEM to be produced (at the same time as the KEM) and the combined KEM/DEM ciphertext to be sent as one packet.

[Bar+20a] Barnes et al., *The Messaging Layer Security (MLS) Protocol*

[Res+20] Rescorla et al., *TLS Encrypted Client Hello*

[Bar+22] Barnes et al., *Hybrid Public Key Encryption*

[Bar+20b] Barnes et al., *Hybrid Public Key Encryption*

[CS03] Cramer and Shoup, "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack"

[Zhe97] Zheng, "Digital Signcryption or How to Achieve Cost(Signature & Encryption) $\ll$ Cost(Signature) + Cost(Encryption)"

[Nat13] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*

[LHT16] Langley et al., *Elliptic Curves for Security*

More interestingly, at least for MLS, authenticating the sender of an HPKE ciphertext (based on their public keys) is clearly also a useful property. (For the ESNI application things are less clear.[1])

In a bit more detail, MLS is already equipped with a notion of a PKI involving public keys bound to long-term identities of parties (as described in [Oma+20]). To invite a new member to an existing MLS protocol session the inviter must send an HPKE ciphertext to the new member. In line with MLS's strong authentication goals, the new member is expected to be able to cryptographically validate the (supposed) identity of the sender of such ciphertexts.

Currently, MLS calls for the HPKE ciphertext to be produced using HPKE's basic mode $\mathsf{HPKE}_{\mathsf{Base}}$ and the resulting ciphertext to be signed by the inviter using a digital signature scheme (either ECDSA or EdDSA). However, an alternative approach to achieve the same ends could be to directly use HPKE in its authenticated mode $\mathsf{HPKE}_{\mathsf{Auth}}$. This would save on at least 2 modular exponentiations as well as result in packets containing 2 fewer group elements. Reducing computational and communication complexity has been a central focus of the MLS design process as such costs are considered the main hurdles to achieving the MLS's stated goal of supporting extremely large groups. Unfortunately, in our analysis, we discovered that $\mathsf{HPKE}_{\mathsf{Auth}}$ does not authenticate the sender when the receiver's secret key leaked, a key compromise impersonation (KCI) attack (Section 4.5.4). MLS aims to provide strong security in the face of state leakage (which includes KCI attacks), so switching from $\mathsf{HPKE}_{\mathsf{Base}}$ and signatures to $\mathsf{HPKE}_{\mathsf{Auth}}$ would result in a significant security downgrade.

$\mathsf{HPKE}_{\mathsf{Auth}}$ could also be a replacement for the authenticated public-key encryption originally implemented by the NaCl cryptographic library. $\mathsf{HPKE}_{\mathsf{Auth}}$ is safer than the NaCl implementation because, in $\mathsf{HPKE}_{\mathsf{Auth}}$, the shared secret is bound to the intended sender and recipient public keys.

[1] The ESNI RFC calls for a client initiating a TLS connection to send an HPKE ciphertext to the server. Although not as common, TLS can also be used in settings with bi-directional authentication. In particular, clients can use certificates binding their identities to their public key to authenticate themselves to the server. Unfortunately, it is unclear how the server would know, a priori, which public key to use for the client when attempting to decrypt the HPKE ciphertext.

[Oma+20] Omara et al., *The Messaging Layer Security (MLS) Architecture*

### 4.1.1 *Our Contributions*

So far, there has been no formal analysis of the HPKE standard. Unfortunately, due to its many modes, options and features a complete analysis of HPKE from scratch seems rather too ambitious for a single work such as this one. Thus, we are forced to choose our scope more carefully. The basic mode $\mathsf{HPKE}_{\mathsf{Base}}$ (especially using the single-shot API) seems to be a quite standard construction. Therefore, and in light of the above discussion around MLS, we have opted to focus on the more novel authenticated mode in its single-shot API form $\mathsf{HPKE}_{\mathsf{Auth}}$. To this end we make the following contributions.

AUTHENTICATED KEM AND PKE. We begin, in Section 4.5, by introducing *Authenticated Key Encapsulation Mechanisms* (AKEM) and *Authenticated Public Key Encryption* (APKE) schemes, where the syntax of APKE matches that of the single-shot authenticated mode of $\mathsf{HPKE}_{\mathsf{Auth}}$. In terms of security, we define (multi-user) security notions capturing both authenticity and (2 types of) privacy for an AKEM and an APKE. In a bit more detail, both for authenticity and for privacy we consider so called weaker *outsider* and stronger *insider* variants. Intuitively, outsider notions model settings where the adversary is an outside observer. Conversely, insider notions model

TABLE 4.1: Security properties needed to prove Outsider-Auth, Outsider-CCA, and Insider-CCA security of APKE obtained by the AKEM/DEM construction.

| | AKEM | | | AEAD | |
|---|---|---|---|---|---|
| | Outsider-Auth | Outsider-CCA | Insider-CCA | INT-CTXT | IND-CPA |
| $\text{Outsider-Auth}_{\text{APKE}}$ | X | X | | X | |
| $\text{Outsider-CCA}_{\text{APKE}}$ | | X | | X | X |
| $\text{Insider-CCA}_{\text{APKE}}$ | | | X | X | X |

settings where the adversary is somehow directly involved; in particular, even selecting some of the secrets used to produce target ciphertexts. A bit more formally, we call an honestly generated key pair *secure* if the secret key was not (explicitly) leaked to the adversary and *leaked* if it was. A key pair is called *bad* if it was sampled arbitrarily by the adversary. A scheme is outsider-secure if target ciphertexts are secure when produced using secure key pairs. Meanwhile, insider security holds even if one secure *and one bad key pair* are used. For example, insider privacy (Insider-CCA) for AKEM requires that an encapsulated key remains indistinguishable from random despite the encapsulating ciphertext being produced using bad sender keys (but secure receiver keys). Similarly, insider authenticity (Insider-Auth) requires that an adversary cannot produce a valid ciphertext for bad receiver keys as long as the sender keys are secure. In particular, insider authenticity implies (but is strictly stronger than) Key Compromise Impersonation (KCI) security as KCI security only requires authenticity for leaked (but not bad) receiver keys.

Moreover, as an independent contribution we show that for each security notion of an AKEM a (significantly simpler) single-user and single-challenge-query version already implies security for its (more complex but practically relevant) multi-user version. In particular, this provides an easier target for future work on AKEMs, e.g. when building a post-quantum variant of $\text{HPKE}_{\text{Auth}}$.

AKEM/DEM: FROM AKEM TO APKE. Next we turn to the AKEM/DEM construction used in the HPKE standard. We prove a set of composition results each showing a different type of security for the single-shot AKEM/DEM construction depending on which properties the underlying AKEM guarantees. Each of these results also assumes standard security properties for the AEAD (namely IND-CPA and INT-CTXT) and for the key schedule KS (namely pseudo-randomness). In particular, these results are proven in the standard model. Somewhat to our surprise, it turns out that the APKE obtained by the AKEM/DEM construction does not provide insider authenticity (and so, nor does $\text{HPKE}_{\text{Auth}}$ itself). Indeed, we give an attack in Section 4.5.4.

Table 4.1 summarises the AKEM and AEAD properties we use to prove each of the remaining 3 types of security for the AKEM/DEM APKE construction.

THE $\text{HPKE}_{\text{Auth}}$ SCHEME. In Section 4.6 we analyse the generic $\text{HPKE}_{\text{Auth}}$ scheme proposed in the RFC. $\text{HPKE}_{\text{Auth}}$ is an instantiation of the AKEM/DEM paradigm discussed above.

Thus, we first analyse DH-AKEM, the particular AKEM underlying $\text{HPKE}_{\text{Auth}}$. The RFC builds DH-AKEM from a key-derivation function KDF and an underlying generic Diffie-Hellman group. As one of our main results

we show that DH-AKEM provides authenticity and privacy based on the Gap Diffie-Hellman assumption over the underlying group. To show this we model KDF as a random oracle.

Next we consider $\mathsf{HPKE}_{\mathsf{Auth}}$'s key schedule and prove it to be pseudo-random based on pseudo-randomness of its building blocks, the functions Extract and Expand. Similarly, we argue why DH-AKEM's key derivation function KDF can be modelled as a random oracle. Finally, by applying our results about the AKEM/DEM paradigm from the previous sections, we obtain security proofs capturing the privacy and authenticity of $\mathsf{HPKE}_{\mathsf{Auth}}$ as an APKE. Our presentation ends with concrete bounds of $\mathsf{HPKE}_{\mathsf{Auth}}$'s security and their interpretation.

PRACTICE-ORIENTED CRYPTOGRAPHY. Due to the very applied nature of HPKE we have taken care to maximise the practical relevance of our results. All security properties we analyse for $\mathsf{HPKE}_{\mathsf{Auth}}$ are defined directly for a multi-user setting. Further, to help practitioners set sound parameters for their HPKE applications, our results are stated in terms of very fine-grained exact (as opposed to asymptotic) terms. That is, the security loss for each result is bounded as an explicit function of various parameters such as the numbers of key pairs, queries, etc.

Finally, instead of relying on a generic prime-order group to state our underlying security assumptions, we ultimately reduce security to assumptions on each of the concrete elliptic-curve-based instantiations. For the P-256, P-384, and P-521 curves, this is relatively straightforward. However, for Curve25519 and Curve448, this is a less than trivial step as those groups (and their associated Diffie-Hellman functions X25519 and X448) depart significantly from the standard generic group abstraction. To this end we introduce the new abstraction of *nominal groups* which allows us to argue about correctness and security of our schemes over all above-mentioned elliptic curve groups, including Curve25519 and Curve448. (We believe this abstraction has applications well beyond its use in this work.) Ultimately, this approach results in both an additional security loss and the explicit consideration of (potential) new attacks not present for generic groups. In particular, both Curve25519 and Curve448 exhibit similar (but different) idiosyncrasies such as having non-equal but functionally equivalent curve points as well as self-reducibility with non-zero error probability, all of which we take into account in our reductions to the respective underlying assumption.

### 4.1.2 *Proof Techniques*

The results in this work have been demonstrated using a combination of traditional "pen-and-paper" techniques and the automated theorem proving tool CryptoVerif [Bla08], which was already used to verify important practical protocols such as TLS 1.3 [BBK17], Signal [KBB17], and WireGuard, the latter in Chapter 3 of this thesis. CryptoVerif produces game-based proofs: it starts from an initial game provided by the user, which represents the protocol or scheme to prove; it transforms this game step by step using a predefined set of game transformations, until it reaches a game on which the desired security properties can easily be proved from the form of the game. The game transformations are guaranteed to produce computation-

[Bla08] Blanchet, "A Computationally Sound Mechanized Prover for Security Protocols"

[BBK17] Bhargavan et al., "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate"

[KBB17] Kobeissi et al., "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach"

ally indistinguishable games, and either rely on a proof by reduction to a computational assumption or are syntactic transformations (e.g. replace a variable with its value). Using CryptoVerif to prove statements can result in greater confidence in their correctness, especially when the proofs require deriving (otherwise quite tedious) exact bounds on the security loss and/or reasoning about relatively complicated, e.g. multi-instance, security games.

However, CryptoVerif also has its limitations. Fortunately, these can be readily overcome using traditional techniques. The language used to define security statements in CryptoVerif is rather unconventional in the context of cryptography, not to mention (necessarily) very formal and detailed. Together this can make it quite challenging to build an intuitive understanding for a given notion (e.g. to verify that it captures the desired setting). To circumvent this, we present each of our security definitions using the more well-known language of game-based security. Next we map these to corresponding CryptoVerif definitions. Thus, the intuition can be built upon a game-based notion and it remains only to verify the *functional equivalence* of the CryptoVerif instantiation.

CryptoVerif was designed with multi-instance security in mind and so relies on more unconventional multi-instance number theoretic assumptions. However, the simpler a definition (say, for a KEM) the easier it is to demonstrate for a given construction. Similarly, in cryptography we tend to prefer simpler, static, not to mention well-known, number theoretic assumptions so as to build more confidence in them. Consequently, we have augmented the automated proofs with further pen-and-paper proofs reducing multi-instance security notions and assumptions to simpler (and more conventional) single-instance versions.

### 4.1.3 *Related Work*

Hybrid cryptography (of which the AKEM/DEM construction in this work is an example) is a widely used technique for constructing practically efficient asymmetric primitives. In particular, there exist several hybrid PKE-based concrete standards predating HPKE, mostly based on the DHIES scheme of [ABR01] defined over a generic (discrete log) group. When the group is instantiated using elliptic curves the result is often referred to as ECIES (much like the Diffie-Hellman scheme over an elliptic curve group is referred to as ECDH). A description and comparison of the most important such standards can be found in [GM+10]. However, per the HPKE RFC, "All these existing schemes have problems, e.g., because they rely on outdated primitives, lack proofs of IND-CCA2 security, or fail to provide test vectors." Moreover, to the best of our knowledge, none of these standards provide a means for authenticating senders.

[ABR01] Abdalla et al., "The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES"

[GM+10] Gayoso Martínez et al., "A comparison of the standardized versions of ECIES"

The APKE primitive we analyse in this chapter can be viewed as a flavour of signcryption [Zhe97]; a family of primitives intended to efficiently combine signatures and public key encryption. Signcryption literature is substantial and we refer to the textbook [DZ10] for an extensive exposition thereof. We highlight some chapters of particular relevance. Chapters 2 and 3 cover 2-party and multi-party security notions, respectively; both for insider and outsider variants. Chapter 4 of [DZ10] contains several

[Zhe97] Zheng, "Digital Signcryption or How to Achieve Cost(Signature & Encryption) $\ll$ Cost(Signature) + Cost(Encryption)"

[DZ10] Dent and Zheng, *Practical Signcryption*

(Gap)-Diffie-Hellman-based signcryption constructions. Finally, Chapter 7 covers some AKEM security notions and constructions (aka. "signcryption KEM") as well as hybrid signcryption constructions such as the outsider-secure one of [Den05b] and insider-secure one of [Den05a]. In contrast to our work, almost all security notions in [DZ10] forbid honest parties from reusing the same key pair for both sending and receiving (even if sender and receiver keys have identical distribution). [2] Nor is it clear that a scheme satisfying a "key-separated" security notion could be converted into an equally efficient scheme supporting key reuse. The naïve transformation (embedding a sender and receiver key pair into a single reusable key pair) would double key sizes. However, an HPKE public key consists of a *single* group element which can be used simultaneously as a sender and receiver public key.

[Den05b] Dent, "Hybrid Signcryption Schemes with Outsider Security"

[Den05a] Dent, "Hybrid Signcryption Schemes with Insider Security"

[DZ10] Dent and Zheng, *Practical Signcryption*

[2] The only exception we are aware of are the security notions used to analyse 2 bilinear-pairing-based schemes in Sections 5.5 and 5.6 of [DZ10].

Recently, Bellare and Stepanovs analysed the signcryption scheme underlying the iMessage secure messaging protocol [BS20]. Although their security notions allow for key reuse as in our work, they fall outside the outsider/insider taxonomy common in signcryption literature. Instead, they capture an intermediary variant more akin to KCI security.

[BS20] Bellare and Stepanovs, "Security Under Message-Derived Keys: Signcryption in iMessage"

A detailed model of Curve25519 [LHT16] in CryptoVerif was already presented in Chapter 3 of this thesis; such a model was needed for the proof of the WireGuard protocol. In this chapter, we present a more generic model that allows us to deal not only with Curve25519 but also with prime order groups such as NIST curves [Nat13] in a single model. Moreover, we handle rerandomisation of curve elements, which was not taken into account in Chapter 3.

[LHT16] Langley et al., *Elliptic Curves for Security*

[Nat13] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*

A preliminary version of this work analyses HPKE as a single protocol, not in a modular KEM/DEM setting, it is presented in Section 5.1 of this thesis. The proven theorems are less strong than the ones in this work, e.g. the adversary cannot choose secret keys but only compromise them. However, the analysis covers the single-shot encryption form of all four modes including the secret export API.

## 4.2 PRELIMINARIES

SETS AND ALGORITHMS. We write $h \xleftarrow{\$} \mathcal{S}$ to denote that the variable $h$ is uniformly sampled from the finite set $\mathcal{S}$. For integers $N, M \in \mathbb{N}$, we define $[N, M] := \{N, N+1, \ldots, M\}$ (which is the empty set for $M < N$), $[N] := [1, N]$ and $[N]_0 := [0, N]$. The statistical distance between two random variables $U$ and $V$ having a common domain $\mathcal{U}$ is defined as $\Delta[U, V] = \sum_{u \in \mathcal{U}} |\Pr[U = u] - \Pr[V = u]|$. The notation $[\![B]\!]$, where $B$ is a boolean statement, evaluates to 1 if the statement is true and 0 otherwise.

We use uppercase letters $\mathcal{A}, \mathcal{B}$ to denote algorithms. Unless otherwise stated, algorithms are probabilistic, and we write $(y_1, \ldots) \xleftarrow{\$} \mathcal{A}(x_1, \ldots)$ to denote that $\mathcal{A}$ returns $(y_1, \ldots)$ when run on input $(x_1, \ldots)$. We write $\mathcal{A}^{\mathcal{B}}$ to denote that $\mathcal{A}$ has oracle access to $\mathcal{B}$ during its execution. For a randomised algorithm $\mathcal{A}$, we use the notation $y \in \mathcal{A}(x)$ to denote that $y$ is a possible output of $\mathcal{A}$ on input $x$. We denote the running time of an algorithm $\mathcal{A}$ by $t_{\mathcal{A}}$.

SECURITY GAMES. We use standard code-based security games [BR04]. A *game* $\mathrm{G}$ is a probability experiment in which an adversary $\mathcal{A}$ interacts with an implicit challenger that answers oracle queries issued by $\mathcal{A}$. The game $\mathrm{G}$ has one *main procedure* and an arbitrary amount of additional *oracle procedures* which describe how these oracle queries are answered. We denote the (binary) output $b$ of game $\mathrm{G}$ between a challenger and an adversary $\mathcal{A}$ as $\mathrm{G}^{\mathcal{A}} \Rightarrow b$. $\mathcal{A}$ is said to *win* $\mathrm{G}$ if $\mathrm{G}^{\mathcal{A}} \Rightarrow 1$. Unless otherwise stated, the randomness in the probability term $\Pr[\mathrm{G}^{\mathcal{A}} \Rightarrow 1]$ is over all the random coins in game $\mathrm{G}$.

[BR04] Bellare and Rogaway, *Code-Based Game-Playing Proofs and the Security of Triple Encryption*

## 4.3 STANDARD CRYPTOGRAPHIC DEFINITIONS

A keyed function $F$ with a finite key space $\mathcal{K}$ and a finite output range $\mathcal{R}$ is a function $F : \mathcal{K} \times \{0,1\}^* \to \mathcal{R}$.

**Definition 4.1** (Multi-Key Pseudorandom Function)**.** The $(n_k, q_{\mathsf{PRF}})$-PRF advantage of an adversary $\mathcal{A}$ against a keyed function $F$ with finite key space $\mathcal{K}$ and finite range $\mathcal{R}$ is defined as

$$\mathsf{Adv}^{(n_k,q_{\mathsf{PRF}})\text{-}\mathsf{PRF}}_{F,\mathcal{A}} := \left| \Pr[\mathcal{A}^{f_1(\cdot),\ldots,f_{n_k}(\cdot)}] - \Pr_{k_1,\ldots,k_{n_k} \xleftarrow{\$} \mathcal{K}}[\mathcal{A}^{F(k_1,\cdot),\ldots,F(k_{n_k},\cdot)}] \right| ,$$

where $f_i : \{0,1\}^* \to \mathcal{R}$ for $i \in [n_k]$ are perfect random functions and $\mathcal{A}$ makes at most $q_{\mathsf{PRF}}$ queries in total to the oracles $f_i$, resp. $F(k_i, \cdot)$.

**Definition 4.2** (Collision Resistance)**.** Let $\mathcal{H}$ be a family of hash functions from $\{0,1\}^*$ to the finite range $\mathcal{R}$. We define the advantage of an adversary $\mathcal{A}$ against collision resistance of $\mathcal{H}$ as

$$\mathsf{Adv}^{\mathsf{CR}}_{\mathcal{H},\mathcal{A}} := \Pr[\mathsf{H} \xleftarrow{\$} \mathcal{H}; x_1, x_2 \xleftarrow{\$} \mathcal{A}^{\mathsf{H}} : \mathsf{H}(x_1) = \mathsf{H}(x_2) \wedge x_1 \neq x_2] .$$

We now define (nonce-based) Authenticated Encryption with Associated Data.

**Definition 4.3** (AEAD)**.** A nonce-based authenticated encryption scheme with associated data and key space $\mathcal{K}$ consists of the following two algorithms:

- Deterministic algorithm AEAD.Enc takes as input a key $k \in \mathcal{K}$, a message $m$, associated data $\mathrm{aad}$ and a nonce $\mathrm{nonce}$ and outputs a ciphertext $c$.
- Deterministic algorithm AEAD.Dec takes as input a key $k \in \mathcal{K}$, a ciphertext $c$, associated data $\mathrm{aad}$ and a nonce $\mathrm{nonce}$ and outputs a message $m$ or the failure symbol $\perp$.

We require that for all $\mathrm{aad} \in \{0,1\}^*, m \in \{0,1\}^*, \mathrm{nonce} \in \{0,1\}^{N_n}$

$$\Pr_{k \xleftarrow{\$} \mathcal{K}}[\mathsf{AEAD.Dec}(k, \mathsf{AEAD.Enc}(k, m, \mathrm{aad}, \mathrm{nonce}), \mathrm{aad}, \mathrm{nonce}) \neq \perp] = 1 ,$$

where $N_n$ is the length of the nonce in bits.

We define the multi-key security games $n_k$-IND-CPA$_\ell$ and $n_k$-IND-CPA$_r$ in Listing 4.1 and $(n_k, q_d)$-INT-CTXT$_\ell$ and $(n_k, q_d)$-INT-CTXT$_r$ in Listing 4.2. The advantage of an adversary $\mathcal{A}$ is

$$\begin{aligned}\mathsf{Adv}^{n_k\text{-IND-CPA}}_{\mathcal{A},\mathsf{AEAD}} &:= \big|\Pr[n_k\text{-IND-CPA}_\ell(\mathcal{A}) \Rightarrow 1] \\ &\quad - \Pr[n_k\text{-IND-CPA}_r(\mathcal{A}) \Rightarrow 1]\big| \,, \\ \mathsf{Adv}^{(n_k,q_d)\text{-INT-CTXT}}_{\mathcal{A},\mathsf{AEAD}} &:= \big|\Pr[(n_k,q_d)\text{-INT-CTXT}_\ell(\mathcal{A}) \Rightarrow 1] \\ &\quad - \Pr[(n_k,q_d)\text{-INT-CTXT}_r(\mathcal{A}) \Rightarrow 1]\big| \,.\end{aligned}$$

Here, we define the IND-CPA and INT-CTXT notions as indistinguishability properties between a left game $G_\ell$ and a right game $G_r$, since this is required by CryptoVerif. In order to use such assumptions, CryptoVerif automatically recognizes when a game corresponds to an adversary interacting with $G_\ell$, and it replaces $G_\ell$ with $G_r$ in that game. Moreover, CryptoVerif requires the games $G_\ell$ and $G_r$ to be formulated in a multi-key setting. That allows CryptoVerif to apply the assumption directly in case the scheme is used with several keys, without having to do a hybrid argument itself. (CryptoVerif infers the multi-key assumption automatically from a single-key assumption only in very simple cases.) Also, we allow only one query to the ENC oracle, where the experiment chooses nonces randomly. It is easy to see that these notions are implied by multi-key definitions of IND-CPA and INT-CTXT where the adversary can choose (non-repeating) nonces.

Listing 4.1: Games $n_k$-IND-CPA$_\ell$ and $n_k$-IND-CPA$_r$ for AEAD. Adversary $\mathcal{A}$ makes at most one query per key to ENC.

$n_k$-IND-CPA$_\ell$ and $n_k$-IND-CPA$_r$

```
01 for i ∈ [n_k]
02     k_i ←$ K
03     nonce_i ←$ {0,1}^{N_n}
04 b ←$ A^{Enc}
05 return b
```

Oracle $\textsc{Enc}(i, m, \text{aad})$

```
06 c ← AEAD.Enc(k_i, m, aad, nonce_i)
07 c ← AEAD.Enc(k_i, 0^{|m|}, aad, nonce_i)
08 return (c, nonce_i)
```

Listing 4.2: Games $(n_k, q_d)$-INT-CTXT$_\ell$ and $(n_k, q_d)$-INT-CTXT$_r$ for AEAD. Adversary $\mathcal{A}$ makes at most one query per key to ENC and at most $q_d$ queries in total to DEC.

$(n_k, q_d)$-INT-CTXT$_\ell$ and $(n_k, q_d)$-INT-CTXT$_r$

```
01 for i ∈ [n_k]
02     k_i ←$ K
03     nonce_i ←$ {0,1}^{N_n}
04 E ← ∅
05 b ←$ A^{Enc,Dec}
06 return b
```

Oracle $\textsc{Enc}(i, m, \text{aad})$

```
07 c ← AEAD.Enc(k_i, m, aad, nonce_i)
08 E ← E ∪ {i, m, c, aad}
09 return (c, nonce_i)
```

Oracle $\textsc{Dec}(i, c, \text{aad})$

```
10 m ← AEAD.Dec(k_i, c, aad, nonce_i)
11 if ∃m' : (i, m', c, aad) ∈ E
12     m ← m'
13 else
14     m ← ⊥
15 return m
```

## 4.4 Elliptic Curves

In this section we introduce the elliptic curves relevant for the HPKE standard, P-256, P-384, P-521 [Nat13], Curve25519 and Curve448 [LHT16], together with relevant security assumptions.

[Nat13] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*

[LHT16] Langley et al., *Elliptic Curves for Security*

### 4.4.1 *Nominal Groups*

We first define *nominal groups*, a general abstract model of elliptic curves, and then show how we instantiate it for each of the above-mentioned curves.

**Definition 4.4.** A nominal group $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \exp)$ consists of an efficiently recognizable finite set of elements $\mathcal{G}$ (also called "group elements"), a base element $g \in \mathcal{G}$, a prime $p$, a finite set of honest exponents $\mathcal{E}_H \subset \mathbb{Z}$, a finite set of exponents $\mathcal{E}_U \subset \mathbb{Z} \setminus p\mathbb{Z}$, and an efficiently computable exponentiation function $\exp : \mathcal{G} \times \mathbb{Z} \to \mathcal{G}$, where we write $X^y$ for $\exp(X, y)$. The exponentiation function is required to have the following properties:

1. $(X^y)^z = X^{yz}$ for all $X \in \mathcal{G}$, $y, z \in \mathbb{Z}$;
2. the function $\phi$ defined by $\phi(x) = g^x$ is a bijection from $\mathcal{E}_U$ to $\{g^x \mid x \in [1, p-1]\}$.

A nominal group is said to be *rerandomisable* when:

3. $g^{x+py} = g^x$ for all $x, y \in \mathbb{Z}$;
4. for all $y \in \mathcal{E}_U$, the function $\phi_y$ defined by $\phi_y(x) = g^{xy}$ is a bijection from $\mathcal{E}_U$ to $\{g^x \mid x \in [1, p-1]\}$.

We remark that even though $\mathcal{G}$ is called the set of (group) elements, it is not required to form a group. Property 2 guarantees that the discrete logarithm is unique in the set $\mathcal{E}_U$. (The index $U$ in $\mathcal{E}_U$ stands for unique.) It is needed to define the DH oracle in Definitions 4.5 and 4.6.

For a nominal group $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \exp)$, we let $\mathrm{D}_H$ be the distribution of honestly generated exponents, that is, the uniform distribution on $\mathcal{E}_H$. Let $\mathrm{D}_U$ be the uniform distribution on $\mathcal{E}_U$. Depending on the choice of $\mathcal{E}_H$ and $\mathcal{E}_U$, these distributions may differ. We define the two statistical parameters

$$\Delta_{\mathcal{N}} := \Delta[\mathrm{D}_H, \mathrm{D}_U], \quad \text{and} \quad P_{\mathcal{N}} = \max_{Y \in \mathcal{G}} \Pr_{x \xleftarrow{\$} \mathcal{E}_H} [Y = g^x].$$

We summarise the expected security level and the concrete upper bounds for $\Delta_{\mathcal{N}}$ and $P_{\mathcal{N}}$ in Table 4.2 of Section 4.6.3 and compute them below.

Prime-Order Groups. The simplest example of a rerandomisable nominal group is when $\mathcal{G} = \mathbb{G}$ is a group of prime order $p$ with generator $g$, exp is defined via the usual scalar multiplication on $\mathbb{G}$, and $\mathcal{E}_H = \mathcal{E}_U = [1, p-1]$.

Since $\mathcal{E}_U = [1, p-1]$, the image of $\phi$ is obviously $\{g^x \mid x \in [1, p-1]\}$. Furthermore, $\phi(x) = \phi(x')$ if and only if $x \equiv x' \mod p$, so $\phi$ is injective on $\mathcal{E}_U$.

For all $x, y \in \mathcal{E}_U$, there exists $x' \in [1, p-1]$ such that $xy \equiv x' \mod p$ (since $x$ and $y$, and so $xy$, are prime to $p$). Since we work in a group of order $p$, $\phi_y(x) = g^{xy} = g^{x'}$, so the image of $\phi_y$ is included in $\{g^{x'} \mid x' \in [1, p-1]\}$. Moreover, for all $y \in \mathcal{E}_U$ and $x' \in [1, p-1]$, there exists $x \in \mathcal{E}_U$ such that

$xy \equiv x' \mod p$, so $\phi_y(x) = g^{x'}$, so the image of $\phi_y$ is $\{g^{x'} \mid x' \in [1, p-1]\}$. Finally, for $y \in \mathcal{E}_U$, $\phi_y(x) = \phi_y(x')$ if and only if $xy \equiv x'y \mod p$, if and only if $x \equiv x' \mod p$ so $\phi_y$ is injective on $\mathcal{E}_U$.

The two distributions $\mathrm{D}_H$ and $\mathrm{D}_U$ are identical, so $\Delta_{\mathcal{N}} = 0$. Since all elements have the same probability, we have $P_{\mathcal{N}} = 1/(p-1)$. The NIST curves P-256, P-384, and P-521 [Nat13] are examples of prime-order groups.

[Nat13] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*

CURVE25519 AND CURVE448. We now show that both Curve25519 and Curve448 [LHT16] can also be seen as rerandomisable nominal groups. They are elliptic curves defined by equations of the form $Y^2 = X^3 + AX^2 + X$ in the field $\mathbb{F}_q$ for a large prime $q$. The curve points are represented only by their $X$ coordinate. When $X^3 + AX^2 + X$ is a square $Y^2$, $X$ represents the curve point $(X, Y)$ or $(X, -Y)$. When $X^3 + AX^2 + X$ is not a square, $X$ does not represent a point on the curve, but on its quadratic twist. The curve is a group of cardinal $kp$ and the twist is a group of cardinal $k'p'$, where $p$ and $p'$ are large primes and $k$ and $k'$ are small integers. For Curve25519, $q = 2^{255} - 19$, $k = 8$, $k' = 4$, $p = 2^{252} + \delta$, $p' = 2^{253} - 9 - 2\delta$ with $0 < \delta < 2^{125}$. For Curve448, $q = 2^{448} - 2^{224} - 1$, $k = k' = 4$, $p = 2^{446} - 2^{223} - \delta$, $p' = 2^{446} + \delta$ with $0 < \delta < 2^{220}$. The base point $Q_0$ is an element of the curve, of order $p$, which generates a subgroup $\mathbb{G}_s$ of the curve. The set of elements $\mathcal{G}$ is the set of bitstrings of 32 bytes for Curve25519, of 56 bytes for Curve448.

[LHT16] Langley et al., *Elliptic Curves for Security*

The exponentiation function is specified as follows, using [Ber06, Theorem 2.1]: We consider the elliptic curve $E(\mathbb{F}_{q^2})$ defined by the equation $Y^2 = X^3 + AX^2 + X$ in a quadratic extension $\mathbb{F}_{q^2}$ of $\mathbb{F}_q$. We define $X_0 : E(\mathbb{F}_{q^2}) \to \mathbb{F}_{q^2}$ by $X_0(\infty) = 0$ and $X_0(X, Y) = X$. For $X \in \mathbb{F}_q$ and $y$ an integer, we define $y \cdot X \in \mathbb{F}_q$ as $y \cdot X = X_0(yQ_X)$, where $Q_X \in E(\mathbb{F}_{q^2})$ is any of the two elements satisfying $X_0(Q_X) = X$. (It is not hard to verify that this mapping is well-defined.) Elements in $\mathcal{G}$ are mapped to elements of $\mathbb{F}_q$ by the function $\mathsf{decode\_pk} : \mathcal{G} \to \mathbb{F}_q$ and conversely, elements of $\mathbb{F}_q$ are mapped to the group elements by the function $\mathsf{encode\_pk} : \mathbb{F}_q \to \mathcal{G}$, such that $\mathsf{decode\_pk} \circ \mathsf{encode\_pk}$ is the identity. (For Curve25519 we have $\mathsf{decode\_pk}(X) = (X \bmod 2^{255}) \bmod q$, for Curve448 $\mathsf{decode\_pk}(X) = X \bmod q$, and $\mathsf{encode\_pk}(X)$ is the representation of $X$ as an element of $\{0, \ldots, q-1\}$.) Finally, $X^y = \mathsf{encode\_pk}(y \cdot \mathsf{decode\_pk}(X))$.

[Ber06] Bernstein, "Curve25519: New Diffie-Hellman Speed Records"

As required by Definition 4.4, we have $(X^y)^z = X^{yz}$. Indeed,

$$\begin{aligned}(X^y)^z &= \mathsf{encode\_pk}(z \cdot \mathsf{decode\_pk}(\mathsf{encode\_pk}(y \cdot \mathsf{decode\_pk}(X)))) \\ &= \mathsf{encode\_pk}(z \cdot y \cdot \mathsf{decode\_pk}(X)) \\ &= \mathsf{encode\_pk}(yz \cdot \mathsf{decode\_pk}(X)) = X^{yz}.\end{aligned}$$

The base element is $g = \mathsf{encode\_pk}(X_0(Q_0))$. It is easy to check that $g^{x+py} = g^x$, since $Q_0$ is an element of order $p$. The honest exponents are chosen uniformly in the set $\mathcal{E}_H = \{kn \mid n \in [M, N]\}$. For Curve25519, $M = 2^{251}$, $N = 2^{252} - 1$. For Curve448, $M = 2^{445}$, $N = 2^{446} - 1$.

Our exponentiation function is closely related to the function X25519 (resp. X448 for Curve448) as defined in [LHT16], namely $\mathsf{X25519}(y, X) = X^{\mathsf{clamp}(y)}$, where $\mathsf{clamp}(y)$ sets and resets some bits in the bitstring $y$ to make sure that $\mathsf{clamp}(y) \in \mathcal{E}_H$. Instead of clamping secret keys together with exponentiation, we clamp them when we generate them, hence we generate honest secret keys in $\mathcal{E}_H$.

We let $\mathcal{E}_U = \{kn \mid n \in [(p+1)/2, p-1]\}$. We have $\phi(x) = g^x = \mathsf{encode\_pk}(X_0(xQ_0))$, so $\phi(x) = \phi(x')$ if and only if $\mathsf{encode\_pk}(X_0(xQ_0)) = \mathsf{encode\_pk}(X_0(x'Q_0))$, if and only if $X_0(xQ_0) = X_0(x'Q_0)$ since $\mathsf{encode\_pk}$ is injective, if and only if $xQ_0 = x'Q_0$ or $xQ_0 = -x'Q_0$, if and only if $x \equiv x' \mod p$ or $x \equiv -x' \mod p$ since $Q_0$ is an element of order $p$.

Let us show that $\phi$ is injective on $\mathcal{E}_U$. Suppose $x = kn$ and $x' = kn'$ are in $\mathcal{E}_U$. We have $\phi(x) = \phi(x')$ if and only if $kn \equiv kn' \mod p$ or $kn \equiv -kn' \mod p$, if and only if $n \equiv n' \mod p$ or $n \equiv -n' \mod p$ since $k$ is prime to $p$, so invertible modulo $p$. Since $n$ and $n'$ are in $[(p+1)/2, p-1]$, that implies $n = n'$, so $x = x'$, which shows the injectivity of $\phi$.

Let us now show that $\phi(\mathcal{E}_U) = \{g^x \mid x \in [1, p-1]\}$. If $x \in \mathcal{E}_U$, then there exists $x'$ such that $x \equiv x' \mod p$ and $x' \in [1, p-1]$ (because $x$ is prime to $p$), so $\phi(x) = \phi(x') \in \{g^x \mid x \in [1, p-1]\}$. Conversely, let $X' = g^{x'} = \phi(x')$ with $x' \in [1, p-1]$. Computing modulo $p$, we can find $n_0 \in [1, p-1]$ such that $kn_0 \equiv x' \mod p$ (by inverting $k$ modulo $p$). If $n_0 \in [(p+1)/2, p-1]$, we let $n = n_0$. Otherwise, $n_0 \in [1, (p-1)/2]$, and we let $n = p - n_0$. In all cases, $n \in [(p+1)/2, p-1]$ and $kn \equiv x' \mod p$ or $kn \equiv -x' \mod p$, so $\phi(kn) = \phi(x') = X'$. Therefore, $\phi$ is a bijection from $\mathcal{E}_U$ to $\{g^x \mid x \in [1, p-1]\}$.

Similarly, for $y \in \mathcal{E}_U$, $\phi_y(x) = \phi_y(x')$ if and only if $xy \equiv x'y \mod p$ or $xy \equiv -x'y \mod p$ if and only if $x \equiv x' \mod p$ or $x \equiv -x' \mod p$ since $y$ is prime to $p$. Like for $\phi$, that shows the injectivity of $\phi_y$ on $\mathcal{E}_U$. Let us now show that $\phi_y(\mathcal{E}_U) = \{g^x \mid x \in [1, p-1]\}$. If $x \in \mathcal{E}_U$, then there exists $x'$ such that $xy \equiv x' \mod p$ and $x' \in [1, p-1]$ (because $x$ and $y$ are prime to $p$), so $\phi_y(x) = g^{x'} \in \{g^x \mid x \in [1, p-1]\}$. Conversely, let $X' = g^{x'}$ with $x' \in [1, p-1]$. Computing modulo $p$, we can find $n_0 \in [1, p-1]$ such that $ykn_0 \equiv x' \mod p$ (by inverting $k$ and $y$ modulo $p$). If $n_0 \in [(p+1)/2, p-1]$, we let $n = n_0$. Otherwise, $n_0 \in [1, (p-1)/2]$, and we let $n = p - n_0$. In all cases, $n \in [(p+1)/2, p-1]$ and $ykn \equiv x' \mod p$ or $ykn \equiv -x' \mod p$, so $\phi_y(kn) = g^{x'} = X'$. Therefore, $\phi_y$ is a bijection from $\mathcal{E}_U$ to $\{g^x \mid x \in [1, p-1]\}$.

**Lemma 4.1.** *For Curve25519, $\Delta_{\mathcal{N}} < 2^{-126}$ and $P_{\mathcal{N}} = 2^{-250}$, and for Curve448, $\Delta_{\mathcal{N}} < 2^{-221}$ and $P_{\mathcal{N}} = 2^{-444}$.*

*Proof.* We have

$$\Delta_{\mathcal{N}} = \Delta[\mathrm{D}_H, \mathrm{D}_U] = \frac{1}{2} \sum_{x \in \mathbb{Z}} |\Pr_{\mathrm{D}_U}(x) - \Pr_{\mathrm{D}_H}(x)| .$$

For Curve25519, we have $M < (p+1)/2 < N < p-1$. The exponents $kn$ for $n \in [M, (p-1)/2]$ each have probability 0 in $\mathrm{D}_U$ and $1/2^{251}$ in $\mathrm{D}_H$. The exponents $kn$ for $n \in [(p+1)/2, N]$ each have probability $2/(p-1)$ in $\mathrm{D}_U$ and $1/2^{251}$ in $\mathrm{D}_H$. The exponents $kn \in [N+1, p-1]$ each have probability $2/(p-1)$ in $\mathrm{D}_U$ and 0 in $\mathrm{D}_H$. Then

$$\begin{aligned} 2\Delta_{\mathcal{N}} = &\left(\frac{p-1}{2} - M + 1\right) \times \frac{1}{2^{251}} + \left(N - \frac{p+1}{2} + 1\right) \times \left|\frac{2}{p-1} - \frac{1}{2^{251}}\right| \\ &+ (p - 1 - (N+1) + 1) \times \frac{2}{p-1} . \end{aligned}$$

Since $1/(p-1) < 2^{-252}$, a straightforward computation shows that

$$\Delta_{\mathcal{N}} < 2^{-126}.$$

For Curve448, we have $(p+1)/2 < M < p-1 < N$. The exponents $kn$ for $n \in [(p+1)/2, M-1]$ each have probability $2/(p-1)$ in $\mathrm{D}_U$ and 0 in $\mathrm{D}_H$. The exponents $kn$ for $n \in [M, p-1]$ each have probability $2/(p-1)$ in $\mathrm{D}_U$ and $1/2^{445}$ in $\mathrm{D}_H$. The exponents $kn$ for $n \in [p, N]$ have probability 0 in $\mathrm{D}_U$ and $1/2^{445}$ in $\mathrm{D}_H$. Then

$$2\Delta_{\mathcal{N}} = \left(M - 1 - \frac{p+1}{2} + 1\right) \times \frac{2}{p-1} + (p-1-M+1) \times \left|\frac{2}{p-1} - \frac{1}{2^{445}}\right| + (N-p+1) \times \frac{1}{2^{445}}.$$

Since $2/(p-1) > 2^{-445}$, a straightforward computation shows that

$$\Delta_{\mathcal{N}} < 2^{-221}.$$

Some elements $g^x$ are generated from 2 exponents $x = kn$ for $n \in [M, N]$ (for Curve25519, the exponents $kn$ for $n \in [M, (p+1)/2-1]$ yield the same elements as those for $n \in [(p+1)/2, p-M]$; for Curve448, the exponents $kn$ for $n \in [p+1, N]$ yield the same elements as those for $n \in [2p-N, p-1]$), so they have probability $\frac{2}{N-M+1}$, and all other elements are generated from one exponent $kn$ with $n \in [M, N]$, so they have probability $\frac{1}{N-M+1}$. Therefore, $P_{\mathcal{N}} = \frac{2}{N-M+1}$, which yields $P_{\mathcal{N}} = 2^{-250}$ for Curve25519 and $P_{\mathcal{N}} = 2^{-444}$ for Curve448. □

### 4.4.2 *Diffie-Hellman Assumptions*

Let us first recall the Gap Diffie-Hellman and Square Gap Diffie-Hellman assumptions, as in this chapter, we use slightly different definitions than for the work on WireGuard (cf. Definition 3.1). We adapt them to the setting of a nominal group $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \exp)$ of the previous section, by allowing elements in $\mathcal{G}$ as arguments of the Diffie-Hellman decision oracle. Moreover, we choose secret keys in $\mathcal{E}_U$, not in $\mathcal{E}_H$, as it guarantees that the secret key $p$, or equivalently 0, is never chosen, which helps in the following theorems.

**Definition 4.5** (Gap Diffie-Hellman (GDH) Problem)**.** We define the advantage function of an adversary $\mathcal{A}$ against the Gap Diffie-Hellman problem over nominal group $\mathcal{N}$ as

$$\mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{A},\mathcal{N}} := \Pr_{x,y \xleftarrow{\$} \mathcal{E}_U} [Z = g^{xy} \mid Z \xleftarrow{\$} \mathcal{A}^{\mathrm{DH}}(g^x, g^y)]$$

where DH is a decision oracle that on input $(g^{\hat{x}}, Y, Z)$ with $\hat{x} \in \mathcal{E}_U$ and $Y, Z \in \mathcal{G}$, returns 1 iff $Y^{\hat{x}} = Z$ and 0 otherwise.

Since $\phi$ is injective on $\mathcal{E}_U$ by Property 2 of Definition 4.4, the value of $\hat{x} \in \mathcal{E}_U$ is unambiguously defined by $g^{\hat{x}}$.

**Definition 4.6** (Square Gap Diffie-Hellman (sqGDH) Problem)**.** We define the advantage function of an adversary $\mathcal{A}$ against the Square Gap Diffie-Hellman problem over nominal group $\mathcal{N}$ as

$$\mathsf{Adv}^{\mathsf{sqGDH}}_{\mathcal{A},\mathcal{N}} := \Pr_{x \xleftarrow{\$} \mathcal{E}_U} \left[Z = g^{x^2} \mid Z \xleftarrow{\$} \mathcal{A}^{\mathrm{DH}}(g^x)\right]$$

where DH is a decision oracle that on input $(g^{\hat{x}}, Y, Z)$, with $\hat{x} \in \mathcal{E}_U$ and $Y, Z \in \mathcal{G}$, returns 1 iff $Y^{\hat{x}} = Z$ and 0 otherwise.

CryptoVerif cannot use cryptographic assumptions directly in this form: as explained in Section 4.3, it requires assumptions to be formulated as computational indistinguishability axioms between a left game $G_\ell$ and a right game $G_r$, formulated in a multi-key setting. Therefore, we reformulate the Gap Diffie-Hellman assumption to satisfy these requirements, and prove that our formulation is implied by the standard assumption.

We also take into account at this point that secret keys are actually chosen in $\mathcal{E}_H$ rather than in $\mathcal{E}_U$.

**Definition 4.7** (Left-or-Right $(n,m)$-Gap Diffie-Hellman Problem)**.** We define the advantage function of an adversary $\mathcal{A}$ against the left-or-right $(n,m)$-Gap Diffie-Hellman problem over nominal group $\mathcal{N}$ as

$$\mathsf{Adv}^{\mathsf{LoR}\text{-}(n,m)\text{-}\mathsf{GDH}}_{\mathcal{A},\mathcal{N}} := \left| \Pr_{\substack{\forall i \in [n]:\, x_i \xleftarrow{\$} \mathcal{E}_H \\ \forall j \in [m]:\, y_j \xleftarrow{\$} \mathcal{E}_H}} \left[ \mathcal{A}^{\mathrm{DH}_\ell, \mathrm{DH}_x, \mathrm{DH}_y}(g^{x_1}, \ldots, g^{x_n}, g^{y_1}, \ldots, g^{y_m}) \Rightarrow 1 \right] \right.$$
$$\left. - \Pr_{\substack{\forall i \in [n]:\, x_i \xleftarrow{\$} \mathcal{E}_H \\ \forall j \in [m]:\, y_j \xleftarrow{\$} \mathcal{E}_H}} \left[ \mathcal{A}^{\mathrm{DH}_r, \mathrm{DH}_x, \mathrm{DH}_y}(g^{x_1}, \ldots, g^{x_n}, g^{y_1}, \ldots, g^{y_m}) \Rightarrow 1 \right] \right|,$$

where $\mathrm{DH}_x$ is a decision oracle that on input $(i, Y, Z)$ for $i \in [n]$ returns 1 iff $Y^{x_i} = Z$ and 0 otherwise; $\mathrm{DH}_y$ is a decision oracle that on input $(j, Y, Z)$ for $j \in [m]$ returns 1 iff $Y^{y_j} = Z$ and 0 otherwise; $\mathrm{DH}_\ell$ is a decision oracle that on input $(i, j, Z)$ for $i \in [n], j \in [m]$ returns 1 iff $Z = g^{x_i y_j}$ and 0 otherwise; and $\mathrm{DH}_r$ is an oracle that on input $(i, j, Z)$ for $i \in [n], j \in [m]$ always returns 0.

**Definition 4.8** (Left-or-Right $n$-Square Gap Diffie-Hellman Problem)**.** We define the advantage function of an adversary $\mathcal{A}$ against the left-or-right $n$-Square Gap Diffie-Hellman problem over nominal group $\mathcal{N}$ as

$$\mathsf{Adv}^{\mathsf{LoR}\text{-}n\text{-}\mathsf{sqGDH}}_{\mathcal{A},\mathcal{N}} := \left| \Pr_{\forall i \in [n]:\, x_i \xleftarrow{\$} \mathcal{E}_H} \left[ \mathcal{A}^{\mathrm{DH}_\ell, \mathrm{DH}_x}(g^{x_1} \ldots, g^{x_n}) \Rightarrow 1 \right] \right.$$
$$\left. - \Pr_{\forall i \in [n]:\, x_i \xleftarrow{\$} \mathcal{E}_H} \left[ \mathcal{A}^{\mathrm{DH}_r, \mathrm{DH}_x}(g^{x_1}, \ldots, g^{x_n}) \Rightarrow 1 \right] \right|,$$

where $\mathrm{DH}_x$ is a decision oracle that on input $(i, Y, Z)$ for $i \in [n]$ returns 1 iff $Y^{x_i} = Z$ and 0 otherwise; $\mathrm{DH}_\ell$ is a decision oracle that on input $(i, j, Z)$ for $i, j \in [n]$ returns 1 iff $Z = g^{x_i x_j}$ and 0 otherwise; and $\mathrm{DH}_r$ is an oracle that on input $(i, j, Z)$ for $i, j \in [n]$ always returns 0.

**Theorem 4.1** ($\mathsf{GDH} \Rightarrow \mathsf{LoR}\text{-}(n,m)\text{-}\mathsf{GDH}$)**.** *Let $\mathcal{N}$ be a rerandomisable nominal group. For any adversary $\mathcal{A}$ against* $\mathsf{LoR}\text{-}(n,m)\text{-}\mathsf{GDH}$*, there exists an adversary $\mathcal{B}$ against* GDH *such that*

$$\mathsf{Adv}^{\mathsf{LoR}\text{-}(n,m)\text{-}\mathsf{GDH}}_{\mathcal{A},\mathcal{N}} \leq \mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B},\mathcal{N}} + (n+m)\Delta_{\mathcal{N}} ,$$

*$\mathcal{B}$ queries the* DH *oracle as many times as $\mathcal{A}$ queries $\mathrm{DH}_x$, $\mathrm{DH}_y$, $\mathrm{DH}_\ell$, or $\mathrm{DH}_r$, and $t_{\mathcal{B}} \approx t_{\mathcal{A}}$.*

*Proof.* Let $\mathcal{A}$ be an adversary against LoR-$(n,m)$-GDH. We consider a game $G_1 = \forall i \in [n]\colon x_i \xleftarrow{\$} \mathcal{E}_H; \forall j \in [m]\colon y_j \xleftarrow{\$} \mathcal{E}_H; \mathcal{A}^{\mathrm{DH}_b,\mathrm{DH}_x,\mathrm{DH}_y}(g^{x_1},\dots,g^{x_n}, g^{y_1},\dots,g^{y_m})$ where $\mathrm{DH}_b$ is an oracle that on input $(i,j,\hat{Z})$ for $i \in [n]$, $j \in [m]$ raises event BAD iff $\hat{Z} = g^{x_i y_j}$, and always returns 0. We have $\mathsf{Adv}^{\text{LoR-}(n,m)\text{-GDH}}_{\mathcal{A},\mathcal{N}} \leq \Pr[G_1 : \mathrm{BAD}]$.

We define a game $G_2$ that runs as $G_1$ except that the exponents $x_i$ and $y_j$ are chosen in $\mathcal{E}_U$ instead of $\mathcal{E}_H$. The probability of distinguishing one exponent in $\mathcal{E}_U$ from one in $\mathcal{E}_H$ is at most $\Delta_{\mathcal{N}}$ and there are $(n+m)$ such exponents, so the probability of distinguishing $G_1$ from $G_2$ is at most $(n+m)\Delta_{\mathcal{N}}$ and $\mathsf{Adv}^{\text{LoR-}(n,m)\text{-GDH}}_{\mathcal{A},\mathcal{N}} \leq \Pr[G_2 : \mathrm{BAD}] + (n+m)\Delta_{\mathcal{N}}$.

We define a game $G_3$ that runs as $G_2$ except that it defines $X_i = g^{x_i}$ and $Y_j = g^{y_j}$ and runs adversary $\mathcal{A}$ on input $(X_1,\dots,X_n,Y_1,\dots,Y_m)$; $\mathrm{DH}_x(i,\hat{Y},\hat{Z})$ returns $\mathrm{DH}(X_i,\hat{Y},\hat{Z})$, $\mathrm{DH}_y(j,\hat{Y},\hat{Z})$ returns $\mathrm{DH}(Y_j,\hat{Y},\hat{Z})$, $\mathrm{DH}_b(i,j,\hat{Z})$ raises event BAD when $\mathrm{DH}(X_i,Y_j,\hat{Z}) = 1$, and always returns 0, where DH is the oracle of Definition 4.5. In the call to DH inside $\mathrm{DH}_x$, we have $\hat{x} = x_i$ since $g^{x_i} = X_i = g^{\hat{x}}$, $x_i$ and $\hat{x}$ are in $\mathcal{E}_U$, and $\phi$ is injective on $\mathcal{E}_U$ by Property 2 of Definition 4.4. The situation is similar in the other oracles, so $G_3$ is perfectly indistinguishable from $G_2$.

We show how to construct an adversary $\mathcal{B}$ against GDH using random self-reducibility.

Adversary $\mathcal{B}$ inputs $(X,Y)$ and samples $r_i \xleftarrow{\$} \mathcal{E}_U$ for $i \in [n]$ and $s_j \xleftarrow{\$} \mathcal{E}_U$ for $j \in [m]$. It computes $X_i = X^{r_i}$ and $Y_j = Y^{s_j}$ and runs adversary $\mathcal{A}$ on input $(X_1,\dots,X_n,Y_1,\dots,Y_m)$. On query $\mathrm{DH}_x(i,\hat{Y},\hat{Z})$, $\mathcal{B}$ queries $\mathrm{DH}(X_i,\hat{Y},\hat{Z})$ and forwards the answer to $\mathcal{A}$ (like $G_3$). On query $\mathrm{DH}_y(j,\hat{Y},\hat{Z})$, $\mathcal{B}$ queries $\mathrm{DH}(Y_j,\hat{Y},\hat{Z})$ and forwards the answer to $\mathcal{A}$ (like $G_3$). On query $\mathrm{DH}_b(i,j,\hat{Z})$, $\mathcal{B}$ queries $\mathrm{DH}(X_i,Y_j,\hat{Z})$. If the output is 0, $\mathcal{B}$ answers 0. If the output is 1, $\mathcal{B}$ immediately aborts the simulation. We can write $X = g^x$ and $Y = g^y$ with $x,y \in \mathcal{E}_U$. Then $X_i = X^{r_i} = g^{xr_i}$ and $Y_j = Y^{s_j} = g^{ys_j}$. Since DH outputs 1, there exists $\hat{x} \in \mathcal{E}_U$ such that $X_i = g^{\hat{x}}$ and $\hat{Z} = Y_j^{\hat{x}} = g^{\hat{x}ys_j} = X_i^{ys_j} = g^{xyr_is_j}$. $\mathcal{B}$ computes the inverse modulo $p$ of $r_is_j$, which is an integer $t$ such that $r_is_jt \equiv 1 \mod p$, and $Z = \hat{Z}^t = g^{xyr_is_jt} = g^{xy}$ since $g^{a+pb} = g^a$ for all $a,b \in \mathbb{Z}$ (Property 3 of Definition 4.4). $\mathcal{B}$ terminates with output $Z$.

Let us show that $X_i$ in this simulation follows the same distribution as $X_i$ in $G_3$. In $G_3$, since $x_i$ is uniformly distributed in $\mathcal{E}_U$ and $\phi$ is a bijection from $\mathcal{E}_U$ to $\{g^x \mid x \in [1,p-1]\}$ (Property 2 of Definition 4.4), $X_i$ is uniformly distributed in $\{g^x \mid x \in [1,p-1]\}$. In the simulation, we have $X_i = X^{r_i} = g^{xr_i}$. Since $r_i$ is uniformly distributed in $\mathcal{E}_U$ and $\phi_x$ is a bijection from $\mathcal{E}_U$ to $\{g^x \mid x \in [1,p-1]\}$ (Property 4 of Definition 4.4), $X_i$ is also uniformly distributed in $\{g^x \mid x \in [1,p-1]\}$. The situation is similar for the keys $Y_j$. Therefore, the simulation perfectly simulates $G_3$, and $G_3$ raises event BAD if and only if the simulation successfully computes $Z$, so $\Pr[G_3 : \mathrm{BAD}] = \mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B},\mathcal{N}}$, and $\mathsf{Adv}^{\text{LoR-}(n,m)\text{-GDH}}_{\mathcal{A},\mathcal{N}} \leq \mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B},\mathcal{N}} + (n+m)\Delta_{\mathcal{N}}$.

The rerandomisation performed here generalizes the one defined for Curve25519 and Curve448 in [BAC19] to a rerandomisable nominal group. □

[BAC19] Barnes et al., *Homomorphic Multiplication for X25519 and X448*

**Theorem 4.2** (sqGDH ⇒ LoR-$n$-sqGDH)**.** *Let $\mathcal{N}$ be a rerandomisable nominal group. For any adversary $\mathcal{A}$ against* LoR-$n$-sqGDH*, there exists an adver-*

*sary $\mathcal{B}$ against* sqGDH *such that*

$$\mathsf{Adv}^{\mathsf{LoR}\text{-}n\text{-}\mathsf{sqGDH}}_{\mathcal{A},\mathcal{N}} \leq \mathsf{Adv}^{\mathsf{sqGDH}}_{\mathcal{B},\mathcal{N}} + n\Delta_{\mathcal{N}} ,$$

*$\mathcal{B}$ queries the* DH *oracle as many times as $\mathcal{A}$ queries* $\mathrm{DH}_x$, $\mathrm{DH}_\ell$, *or* $\mathrm{DH}_r$, *and $t_{\mathcal{B}} \approx t_{\mathcal{A}}$.*

*Proof.* The proof is similar to the proof of Theorem 4.1. Let $\mathcal{A}$ be an adversary against LoR-$n$-sqGDH. We consider a game $G_1 = \forall i \in [n]\colon x_i \xleftarrow{\$} \mathcal{E}_H; \mathcal{A}^{\mathrm{DH}_b,\mathrm{DH}_x}(g^{x_1},\ldots,g^{x_n})$ where $\mathrm{DH}_b$ is an oracle that on input $(i,j,\hat{Z})$ for $i,j \in [n]$ raises event BAD iff $\hat{Z} = g^{x_i x_j}$, and always returns 0. We have $\mathsf{Adv}^{\mathsf{LoR}\text{-}n\text{-}\mathsf{sqGDH}}_{\mathcal{A},\mathcal{N}} \leq \Pr[G_1 : \mathsf{BAD}]$.

We define a game $G_2$ that runs as $G_1$ except that the exponents $x_i$ are chosen in $\mathcal{E}_U$ instead of $\mathcal{E}_H$. The probability of distinguishing one exponent in $\mathcal{E}_U$ from one in $\mathcal{E}_H$ is at most $\Delta_{\mathcal{N}}$ and there are $n$ such exponents, so the probability of distinguishing $G_1$ from $G_2$ is at most $n\Delta_{\mathcal{N}}$ and $\mathsf{Adv}^{\mathsf{LoR}\text{-}n\text{-}\mathsf{sqGDH}}_{\mathcal{A},\mathcal{N}} \leq \Pr[G_2 : \mathsf{BAD}] + n\Delta_{\mathcal{N}}$.

We define a game $G_3$ that runs as $G_2$ except that it defines $X_i = g^{x_i}$ and runs adversary $\mathcal{A}$ on input $(X_1,\ldots,X_n)$; $\mathrm{DH}_x(i,\hat{Y},\hat{Z})$ returns $\mathrm{DH}(X_i,\hat{Y},\hat{Z})$, $\mathrm{DH}_b(i,j,\hat{Z})$ raises event BAD when $\mathrm{DH}(X_i,X_j,\hat{Z}) = 1$, and always returns 0, where DH is the oracle of Definition 4.6. Like in Theorem 4.1, $G_3$ is perfectly indistinguishable from $G_2$.

We show how to construct an adversary $\mathcal{B}$ against sqGDH.

Adversary $\mathcal{B}$ inputs $X$ and samples $r_i \xleftarrow{\$} \mathcal{E}_U$ for $i \in [n]$. It computes $X_i = X^{r_i}$ and runs adversary $\mathcal{A}$ on input $(X_1,\ldots,X_n)$. On query $\mathrm{DH}_x(i,\hat{Y},\hat{Z})$, $\mathcal{B}$ queries $\mathrm{DH}(X_i,\hat{Y},\hat{Z})$ and forwards the answer to $\mathcal{A}$ (like $G_3$). On query $\mathrm{DH}_b(i,j,\hat{Z})$, $\mathcal{B}$ queries $\mathrm{DH}(X_i,X_j,\hat{Z})$. If the output is 0, $\mathcal{B}$ answers 0. If the output is 1, $\mathcal{B}$ immediately aborts the simulation. We can write $X = g^x$ with $x \in \mathcal{E}_U$. Then $X_i = X^{r_i} = g^{xr_i}$ and $X_j = g^{xr_j}$. Since DH outputs 1, there exists $\hat{x} \in \mathcal{E}_U$ such that $X_i = g^{\hat{x}}$ and $\hat{Z} = X_j^{\hat{x}} = g^{\hat{x}xr_j} = X_i^{xr_j} = g^{x^2 r_i r_j}$. $\mathcal{B}$ computes the inverse modulo $p$ of $r_i r_j$, which is an integer $t$ such that $r_i r_j t \equiv 1 \mod p$, and $Z = \hat{Z}^t = g^{x^2 r_i r_j t} = g^{x^2}$ since $g^{a+pb} = g^a$ for all $a,b \in \mathbb{Z}$ (Property 3 of Definition 4.4). $\mathcal{B}$ terminates with output $Z$.

As in the proof of Theorem 4.1, $X_i$ in this simulation follows the same distribution as $X_i$ in $G_3$. Therefore, the simulation perfectly simulates $G_3$, and $G_3$ raises event BAD if and only if the simulation successfully computes $Z$, so $\Pr[G_3 : \mathsf{BAD}] = \mathsf{Adv}^{\mathsf{sqGDH}}_{\mathcal{B},\mathcal{N}}$, and $\mathsf{Adv}^{\mathsf{LoR}\text{-}n\text{-}\mathsf{sqGDH}}_{\mathcal{A},\mathcal{N}} \leq \mathsf{Adv}^{\mathsf{sqGDH}}_{\mathcal{B},\mathcal{N}} + n\Delta_{\mathcal{N}}$. □

IMPLEMENTATION IN CRYPTOVERIF. Definitions in this style for many cryptographic primitives are included in a standard library of cryptographic assumptions in CryptoVerif. As a matter of fact, this library includes a more general variant of the Gap Diffie-Hellman assumption, with corruption oracles and with a decision oracle $\mathrm{DH}(g,X,Y,Z)$, which allows the adversary to choose $g$. In this chapter, we use the definition above as it is sufficient for our proofs.

## 4.5 AUTHENTICATED KEY ENCAPSULATION AND PUBLIC KEY ENCRYPTION

In Section 4.5.1, we introduce notation and security notions for an authenticated key encapsulation mechanism (AKEM), namely Outsider-CCA, Insider-

CCA and Outsider-Auth. In Section 4.5.2, we introduce notation and security notions for authenticated public key encryption (APKE) which follow the ideas of the notions defined for AKEM. Additionally, we define Insider-Auth security.

In Section 4.5.3, we show how to construct an APKE scheme which achieves Outsider-CCA, Insider-CCA and Outsider-Auth, from an AKEM, a PRF, and an AEAD scheme. For Insider-Auth, we give a concrete attack in Section 4.5.4.

### 4.5.1 *Authenticated Key Encapsulation Mechanism*

**Definition 4.9** (AKEM)**.** An authenticated key encapsulation mechanism AKEM consists of three algorithms:

- Gen outputs a key pair $(\mathrm{sk}, \mathrm{pk})$, where $\mathrm{pk}$ defines a key space $\mathcal{K}$.
- AuthEncap takes as input a (sender) secret key $\mathrm{sk}$ and a (receiver) public key $\mathrm{pk}$, and outputs an encapsulation $c$ and a shared secret $K \in \mathcal{K}$.
- Deterministic AuthDecap takes as input a (receiver) secret key $\mathrm{sk}$, a (sender) public key $\mathrm{pk}$, and an encapsulation $c$, and outputs a shared key $K \in \mathcal{K}$.

We require that for all $(\mathrm{sk}_1, \mathrm{pk}_1) \in \mathsf{Gen}, (\mathrm{sk}_2, \mathrm{pk}_2) \in \mathsf{Gen}$,

$$\Pr_{(c,K)\xleftarrow{\$}\mathsf{AuthEncap}(\mathrm{sk}_1,\mathrm{pk}_2)}[\mathsf{AuthDecap}(\mathrm{sk}_2, \mathrm{pk}_1, c) = K] = 1 \,.$$

The two sets of secret and public keys, $\mathcal{SK}$ and $\mathcal{PK}$, are defined via the support of the Gen algorithm as $\mathcal{SK} := \{\mathrm{sk} \mid (\mathrm{sk}, \mathrm{pk}) \in \mathsf{Gen}\}$ and $\mathcal{PK} := \{\mathrm{pk} \mid (\mathrm{sk}, \mathrm{pk}) \in \mathsf{Gen}\}$. We assume that there exists a projection function $\mu : \mathcal{SK} \to \mathcal{PK}$, such that for all $(\mathrm{sk}, \mathrm{pk}) \in \mathsf{Gen}$ it holds that $\mu(\mathrm{sk}) = \mathrm{pk}$. Note that such a function exists without loss of generality by defining $\mathrm{sk}$ to be the randomness $\mathrm{rnd}$ used in the key generation.

Finally, the key collision probability $P_{\mathsf{AKEM}}$ of AKEM is defined as

$$P_{\mathsf{AKEM}} := \max_{\mathrm{pk}\in\mathcal{PK}} \Pr_{(\mathrm{sk}',\mathrm{pk}')\xleftarrow{\$}\mathsf{Gen}}[\mathrm{pk} = \mathrm{pk}'] \,.$$

PRIVACY. The following security notions for privacy aim to capture that KEM ciphertexts issued to an honest and uncompromised receiver are secure, i. e., do not leak any information about the KEM shared key. In Outsider-CCA, the adversary is an outsider, meaning it has no knowledge about the secret keys of honest senders or honest receivers. In Insider-CCA, the adversary is an insider, meaning it gets to choose the secret key of the sender. In both variants, the adversary gets access to encapsulation and decapsulation oracles that it can call multiple times, and in any order.

We define the games $(n, q_e, q_d)$-Outsider-CCA$_\ell$ and $(n, q_e, q_d)$-Outsider-CCA$_r$ in Listing 4.3 and the games $(n, q_e, q_d, q_c)$-Insider-CCA$_\ell$ and $(n, q_e, q_d, q_c)$-Insider-CCA$_r$ in Listing 4.4. The games follow the left-or-right style, as CryptoVerif requires this for assumptions, and we use these notions as assumptions in the composition theorems. In Appendix C.2, we compare the code-based game syntax with the CryptoVerif syntax for Outsider-CCA.

In all games, the adversary has access to a key generation oracle GEN, which allows to create key pairs for up to $n$ users. The adversary will get the public key and an index as identifier for use in other oracles. In the Outsider-CCA games, the adversary has additional access to oracles AENCAP and ADECAP. AENCAP takes as input an index specifying the sender, as well as an arbitrary public key specifying the receiver, and returns a ciphertext and a KEM key. In the left game Outsider-CCA$_\ell$, AENCAP always returns the real KEM key. In the right game Outsider-CCA$_r$, it outputs a uniformly random key if the receiver public key was generated by the experiment. Queries to ADECAP, where the adversary specifies an index for a receiver public key, an arbitrary sender public key and a ciphertext, output a KEM key. In the Outsider-CCA$_r$ game, we use a multiset $\mathcal{E}$ to store queries to AENCAP ($\uplus$ denotes multiset union), which allows us to keep the output of ADECAP consistent. We use a multiset rather than a set here since it may happen that $\mathcal{E}$ contains more than one entry $(\mathrm{pk}, \mathrm{pk}', c, K)$, for the same or for different $K$, in case ciphertexts chosen by AENCAP collide. The instruction **try get** $K$ s. t. $(\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{E}$ **then** computes the multiset $\{K \mid (pk, pk', c, K) \in \mathcal{E}\}$. If that multiset is not empty, it chooses one element of it uniformly at random and stores it in $K$. Then, it executes the code in the **then** branch. If that multiset is empty, the code in the **then** branch is not executed.

In the Insider-CCA games, there is an additional challenge oracle CHALL. The adversary provides an index specifying the receiver and the secret key of the sender, thus taking the role of an insider. CHALL will then output the real KEM key in the Insider-CCA$_\ell$ game, and a uniformly random key in the Insider-CCA$_r$ game. Thus, even if the target ciphertext was produced with a bad sender secret key (and honest receiver public key), the KEM key should be indistinguishable from a random key. AENCAP will always output the real key and the output of ADECAP is kept consistent with challenges using the same notation for the multiset $\mathcal{E}$ as described above.

In all games, the adversary makes at most $n$ queries to GEN, at most $q_e$ queries to oracle AENCAP and at most $q_d$ queries to oracle ADECAP. In the Insider-CCA experiment, it can additionally make at most $q_c$ queries to oracle CHALL. We define the advantage of an adversary $\mathcal{A}$ as

$$\begin{aligned}
\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-CCA}}_{\mathcal{A},\mathsf{AKEM}} &:= \big|\Pr[(n,q_e,q_d)\text{-Outsider-CCA}_\ell(\mathcal{A}) \Rightarrow 1] \\
&\quad - \Pr[(n,q_e,q_d)\text{-Outsider-CCA}_r(\mathcal{A}) \Rightarrow 1]\big| , \\
\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Insider-CCA}}_{\mathcal{A},\mathsf{AKEM}} &:= \big|\Pr[(n,q_e,q_d,q_c)\text{-Insider-CCA}_\ell(\mathcal{A}) \Rightarrow 1] \\
&\quad - \Pr[(n,q_e,q_d,q_c)\text{-Insider-CCA}_r(\mathcal{A}) \Rightarrow 1]\big| .
\end{aligned}$$

AUTHENTICITY. The following Outsider-Auth security notion aims to capture that the adversary is unable to forge a KEM ciphertext to an honest receiver, pretending to come from an honest sender. We do not define an insider security notion because Insider-Auth security is infeasible both for the APKE construction used in HPKE (see Section 4.5.4), and the instantiation of AKEM proposed in HPKE (see end of Section 4.6.1).

We define the games $(n, q_e, q_d)$-Outsider-Auth$_\ell$ and $(n, q_e, q_d)$-Outsider-Auth$_r$ in Listing 4.5. The adversary has access to oracles GEN, AENCAP

Listing 4.3: Games $(n, q_e, q_d)$-Outsider-CCA$_\ell$ and $(n, q_e, q_d)$-Outsider-CCA$_r$ for AKEM. Adversary $\mathcal{A}$ makes at most $n$ queries to GEN, at most $q_e$ queries to AENCAP and at most $q_d$ queries to ADECAP.

$(n, q_e, q_d)$-Outsider-CCA$_\ell$ and
$(n, q_e, q_d)$-Outsider-CCA$_r$

$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$b \xleftarrow{\$} \mathcal{A}^{\text{GEN},\text{AENCAP},\text{ADECAP}}$
**return** $b$

Oracle GEN
$\ell \leftarrow \ell + 1$
$(\text{sk}_\ell, \text{pk}_\ell) \xleftarrow{\$} \text{Gen}$
**return** $(\text{pk}_\ell, \ell)$

Oracle AENCAP$(i \in [\ell], \text{pk})$
$(c, K) \xleftarrow{\$} \text{AuthEncap}(\text{sk}_i, \text{pk})$
**if** $\text{pk} \in \{\text{pk}_1, \ldots, \text{pk}_\ell\}$
  $K \xleftarrow{\$} \mathcal{K}$
  $\mathcal{E} \leftarrow \mathcal{E} \uplus \{(\text{pk}_i, \text{pk}, c, K)\}$
**return** $(c, K)$

Oracle ADECAP$(j \in [\ell], \text{pk}, c)$
**try get** $K$ s. t. $(\text{pk}, \text{pk}_j, c, K) \in \mathcal{E}$
  **then return** $K$
$K \leftarrow \text{AuthDecap}(\text{sk}_j, \text{pk}, c)$
**return** $K$

Listing 4.4: Games $(n, q_e, q_d, q_c)$-Insider-CCA$_\ell$ and $(n, q_e, q_d, q_c)$-Insider-CCA$_r$ for AKEM. Adversary $\mathcal{A}$ makes at most $n$ queries to GEN, at most $q_e$ queries to AENCAP, at most $q_d$ queries to ADECAP and at most $q_c$ queries to CHALL.

$(n, q_e, q_d, q_c)$-Insider-CCA$_\ell$ and
$(n, q_e, q_d, q_c)$-Insider-CCA$_r$

$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$b \xleftarrow{\$} \mathcal{A}^{\text{GEN},\text{AENCAP},\text{ADECAP},\text{CHALL}}$
**return** $b$

Oracle GEN
$\ell \leftarrow \ell + 1$
$(\text{sk}_\ell, \text{pk}_\ell) \xleftarrow{\$} \text{Gen}$
**return** $(\text{pk}_\ell, \ell)$

Oracle AENCAP$(i \in [\ell], \text{pk})$
$(c, K) \xleftarrow{\$} \text{AuthEncap}(\text{sk}_i, \text{pk})$
**return** $(c, K)$

Oracle ADECAP$(j \in [\ell], \text{pk}, c)$
**try get** $K$ s. t. $(\text{pk}, \text{pk}_j, c, K) \in \mathcal{E}$
  **then return** $K$
$K \leftarrow \text{AuthDecap}(\text{sk}_j, \text{pk}, c)$
**return** $K$

Oracle CHALL$(j \in [\ell], \text{sk})$
$(c, K) \xleftarrow{\$} \text{AuthEncap}(\text{sk}, \text{pk}_j)$
$K \xleftarrow{\$} \mathcal{K}$
$\mathcal{E} \leftarrow \mathcal{E} \uplus \{(\mu(\text{sk}), \text{pk}_j, c, K)\}$
**return** $(c, K)$

Listing 4.5: Games $(n, q_e, q_d)$-Outsider-Auth$_\ell$ and $(n, q_e, q_d)$-Outsider-Auth$_r$ for AKEM. Adversary $\mathcal{A}$ makes at most $n$ queries to GEN, at most $q_e$ queries to AENCAP and at most $q_d$ queries to ADECAP.

$(n, q_e, q_d)$-Outsider-Auth$_\ell$ and
$(n, q_e, q_d)$-Outsider-Auth$_r$

$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$b \xleftarrow{\$} \mathcal{A}^{\text{GEN},\text{AENCAP},\text{ADECAP}}$
**return** $b$

Oracle GEN
$\ell \leftarrow \ell + 1$
$(\text{sk}_\ell, \text{pk}_\ell) \xleftarrow{\$} \text{Gen}$
**return** $(\text{pk}_\ell, \ell)$

Oracle AENCAP$(i \in [\ell], \text{pk})$
$(c, K) \xleftarrow{\$} \text{AuthEncap}(\text{sk}_i, \text{pk})$
$\mathcal{E} \leftarrow \mathcal{E} \uplus \{(\text{pk}_i, \text{pk}, c, K)\}$
**return** $(c, K)$

Oracle ADECAP$(j \in [\ell], \text{pk}, c)$
**try get** $K$ s. t. $(\text{pk}, \text{pk}_j, c, K) \in \mathcal{E}$
  **then return** $K$
$K \leftarrow \text{AuthDecap}(\text{sk}_j, \text{pk}, c)$
**if** $\text{pk} \in \{\text{pk}_1, \ldots, \text{pk}_\ell\}$ **and** $K \neq \bot$
  $K \xleftarrow{\$} \mathcal{K}$
  $\mathcal{E} \leftarrow \mathcal{E} \uplus \{(\text{pk}, \text{pk}_j, c, K)\}$
**return** $K$

and ADECAP, where GEN is defined as in the CCA games. AENCAP will always output the real KEM key. ADECAP will output the real key in game Outsider-Auth$_\ell$. In the Outsider-Auth$_r$ game, the adversary (acting as an outsider) will receive a uniformly random key if the receiver public key was generated by the experiment. Thus, the adversary should not be able to distinguish the real KEM key from a random key for two honest users, even if it can come up with the target ciphertext. Again we use the multiset $\mathcal{E}$ to ensure consistency.

The adversary makes at most $n$ queries to GEN, at most $q_e$ queries to oracle AENCAP and at most $q_d$ queries to oracle ADECAP. We define the advantage of an adversary $\mathcal{A}$ as

$$\begin{aligned}\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-Auth}}_{\mathcal{A},\mathsf{AKEM}} := \big|&\Pr[(n,q_e,q_d)\text{-Outsider-Auth}_\ell(\mathcal{A}) \Rightarrow 1] \\ &- \Pr[(n,q_e,q_d)\text{-Outsider-Auth}_r(\mathcal{A}) \Rightarrow 1]\big| \,.\end{aligned}$$

In Appendix C.1, we provide simpler single-user or 2-user versions of these properties, and show that they non-tightly imply the definitions above. These results could be useful to simplify the proof for new AKEMs that could be added to HPKE, such as post-quantum AKEMs. However, because the reduction is not tight, a direct proof of multi-user security may yield better probability bounds. This is the case for our proof of DH-AKEM in Section 4.6.1.

### 4.5.2 *Authenticated Public Key Encryption*

**Definition 4.10** (APKE)**.** An authenticated public key encryption scheme APKE consists of the following three algorithms:

- Gen outputs a key pair $(\mathrm{sk}, \mathrm{pk})$.
- AuthEnc takes as input a (sender) secret key $\mathrm{sk}$, a (receiver) public key $\mathrm{pk}$, a message $m$, associated data $\mathrm{aad}$, a bitstring $\mathrm{info}$, and outputs a ciphertext $c$.
- Deterministic AuthDec takes as input a (receiver) secret key $\mathrm{sk}$, a (sender) public key $\mathrm{pk}$, a ciphertext $c$, associated data $\mathrm{aad}$ and a bitstring $\mathrm{info}$, and outputs a message $m$.

We require that for all messages $m \in \{0,1\}^*, \mathrm{aad} \in \{0,1\}^*, \mathrm{info} \in \{0,1\}^*$,

$$\Pr_{\substack{(\mathrm{sk}_S,\mathrm{pk}_S)\xleftarrow{\$}\mathsf{Gen}\\(\mathrm{sk}_R,\mathrm{pk}_R)\xleftarrow{\$}\mathsf{Gen}}}\left[\begin{array}{l} c \leftarrow \mathsf{AuthEnc}(\mathrm{sk}_S,\mathrm{pk}_R,m,\mathrm{aad},\mathrm{info}),\\ \mathsf{AuthDec}(\mathrm{sk}_R,\mathrm{pk}_S,c,\mathrm{aad},\mathrm{info}) = m\end{array}\right] = 1\,.$$

PRIVACY. We define the games $(n,q_e,q_d,q_c)$-Outsider-CCA and $(n,q_e,q_d,q_c)$-Insider-CCA in Listing 4.6, which follow ideas similar to the games for outsider and insider-secure AKEM. The security notions for APKE use the common style where challenge queries are with respect to a random bit $b$. In particular, the additional challenge oracle CHALL will encrypt either message $m_0$ or $m_1$ provided by the adversary, depending on $b$. Oracles AENC and ADEC will always encrypt and decrypt honestly (except for challenge ciphertexts). In these games, we use a set $\mathcal{E}$ to store queries to

Listing 4.6: Games $(n, q_e, q_d, q_c)$-Outsider-CCA and $(n, q_e, q_d, q_c)$-Insider-CCA for APKE, where $(n, q_e, q_d, q_c)$-Outsider-CCA uses oracle CHALL in the dashed box and $(n, q_e, q_d, q_c)$-Insider-CCA uses oracle CHALL in the solid box. Adversary $\mathcal{A}$ makes at most $n$ queries to GEN, at most $q_e$ queries to AENC, at most $q_d$ queries to ADEC and at most $q_c$ queries to CHALL.

$(n, q_e, q_d, q_c)$-Outsider-CCA and

$(n, q_e, q_d, q_c)$-Insider-CCA

$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$b \xleftarrow{\$} \{0, 1\}$
$b' \xleftarrow{\$} \mathcal{A}^{\text{GEN,AENC,ADEC,CHALL}}$
**return** $[\![b = b']\!]$

Oracle GEN
$\ell \leftarrow \ell + 1$
$(\text{sk}_\ell, \text{pk}_\ell) \xleftarrow{\$} \text{Gen}$
**return** $(\text{pk}_\ell, \ell)$

Oracle ADEC$(j \in [\ell], \text{pk}, c, \text{aad}, \text{info})$
**if** $(\text{pk}, \text{pk}_j, c, \text{aad}, \text{info}) \in \mathcal{E}$
  **return** $\perp$
$m \leftarrow \text{AuthDec}(\text{sk}_j, \text{pk}, c, \text{aad}, \text{info})$
**return** $m$

Oracle AENC$(i \in [\ell], \text{pk}, m, \text{aad}, \text{info})$
$c \xleftarrow{\$} \text{AuthEnc}(\text{sk}_i, \text{pk}, m, \text{aad}, \text{info})$
**return** $c$

Oracle CHALL$(i \in [\ell], j \in [\ell], m_0, m_1, \text{aad}, \text{info})$
**if** $|m_0| \neq |m_1|$ **return** $\perp$
$c \xleftarrow{\$} \text{AuthEnc}(\text{sk}_i, \text{pk}_j, m_b, \text{aad}, \text{info})$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{(\text{pk}_i, \text{pk}_j, c, \text{aad}, \text{info})\}$
**return** $c$

Oracle CHALL$(j \in [\ell], \text{sk}, m_0, m_1, \text{aad}, \text{info})$
**if** $|m_0| \neq |m_1|$ **return** $\perp$
$c \xleftarrow{\$} \text{AuthEnc}(\text{sk}, \text{pk}_j, m_b, \text{aad}, \text{info})$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mu(\text{sk}), \text{pk}_j, c, \text{aad}, \text{info})\}$
**return** $c$

Listing 4.7: Games $(n, q_e, q_d)$-Outsider-Auth and $(n, q_e, q_d)$-Insider-Auth for APKE. Adversary $\mathcal{A}$ makes at most $n$ queries to GEN, at most $q_e$ queries to AENC and at most $q_d$ queries to ADEC.

$(n, q_e, q_d)$-Outsider-Auth
$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$(i^*, j^*, c^*, \text{aad}^*, \text{info}^*) \xleftarrow{\$} \mathcal{A}^{\text{GEN,AENC,ADEC}}$
**return** $[\![(\text{pk}_{i^*}, \text{pk}_{j^*}, c^*, \text{aad}^*, \text{info}^*) \notin \mathcal{E}$
  **and** $\text{AuthDec}(\text{sk}_{j^*}, \text{pk}_{i^*}, c^*, \text{aad}^*, \text{info}^*) \neq \perp]\!]$

$(n, q_e, q_d)$-Insider-Auth
$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$(i^*, \text{sk}, c^*, \text{aad}^*, \text{info}^*) \xleftarrow{\$} \mathcal{A}^{\text{GEN,AENC,ADEC}}$
**return** $[\![(\text{pk}_{i^*}, \mu(\text{sk}), c^*, \text{aad}^*, \text{info}^*) \notin \mathcal{E}$
  **and** $\text{AuthDec}(\text{sk}, \text{pk}_{i^*}, c^*, \text{aad}^*, \text{info}^*) \neq \perp]\!]$

Oracle GEN
$\ell \leftarrow \ell + 1$
$(\text{sk}_\ell, \text{pk}_\ell) \xleftarrow{\$} \text{Gen}$
**return** $(\text{pk}_\ell, \ell)$

Oracle AENC$(i \in [\ell], \text{pk}, m, \text{aad}, \text{info})$
$c \xleftarrow{\$} \text{AuthEnc}(\text{sk}_i, \text{pk}, m, \text{aad}, \text{info})$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{(\text{pk}_i, \text{pk}, c, \text{aad}, \text{info})\}$
**return** $c$

Oracle ADEC$(j \in [\ell], \text{pk}, c, \text{aad}, \text{info})$
$m \leftarrow \text{AuthDec}(\text{sk}_j, \text{pk}, c, \text{aad}, \text{info})$
**return** $m$

Listing 4.8: Authenticated PKE scheme APKE[AKEM, KS, AEAD] construction from AKEM, KS and AEAD, where APKE.Gen = AKEM.Gen.

AuthEnc$(\text{sk}, \text{pk}, m, \text{aad}, \text{info})$
$(c_1, K) \xleftarrow{\$} \text{AuthEncap}(\text{sk}, \text{pk})$
$(k, \text{nonce}) \leftarrow \text{KS}(K, \text{info})$
$c_2 \leftarrow \text{AEAD.Enc}(k, m, \text{aad}, \text{nonce})$
**return** $(c_1, c_2)$

AuthDec$(\text{sk}, \text{pk}, (c_1, c_2), \text{aad}, \text{info})$
$K \leftarrow \text{AuthDecap}(\text{sk}, \text{pk}, c_1)$
$(k, \text{nonce}) \leftarrow \text{KS}(K, \text{info})$
$m \leftarrow \text{AEAD.Dec}(k, c_2, \text{aad}, \text{nonce})$
**return** $m$

CHALL and the adversary is not allowed to trivially decrypt a challenge using ADEC.

In these games, the adversary $\mathcal{A}$ makes at most $n$ queries to GEN, at most $q_e$ queries to oracle AENC, at most $q_d$ queries to oracle ADEC, and at most $q_c$ queries to oracle CHALL. The advantage of $\mathcal{A}$ is

$$\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Outsider-CCA}}_{\mathcal{A},\mathsf{APKE}} := \left|\Pr[(n,q_e,q_d,q_c)\text{-Outsider-CCA}(\mathcal{A}) \Rightarrow 1] - \frac{1}{2}\right| ,$$
$$\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Insider-CCA}}_{\mathcal{A},\mathsf{APKE}} := \left|\Pr[(n,q_e,q_d,q_c)\text{-Insider-CCA}(\mathcal{A}) \Rightarrow 1] - \frac{1}{2}\right| .$$

AUTHENTICITY. Furthermore, we define the games $(n,q_e,q_d)$-Outsider-Auth and $(n,q_e,q_d)$-Insider-Auth in Listing 4.7. The adversary has access to an encryption and decryption oracle and has to come up with a new tuple of ciphertext, associated data and info for any honest receiver secret key (Outsider-Auth) or any (possibly leaked or bad) receiver secret key (Insider-Auth), provided that the sender public key is honest.

In these games, adversary $\mathcal{A}$ makes at most $n$ queries to GEN, at most $q_e$ queries to oracle AENC and at most $q_d$ queries to oracle ADEC. The advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-Auth}}_{\mathcal{A},\mathsf{APKE}} := \Pr[(n,q_e,q_d)\text{-Outsider-Auth}(\mathcal{A}) \Rightarrow 1] ,$$
$$\mathsf{Adv}^{(n,q_e,q_d)\text{-Insider-Auth}}_{\mathcal{A},\mathsf{APKE}} := \Pr[(n,q_e,q_d)\text{-Insider-Auth}(\mathcal{A}) \Rightarrow 1] .$$

### 4.5.3 *From* AKEM *to* APKE

In this section we define and analyse a general transformation that models HPKE's way of constructing APKE from an AKEM (c.f. Definition 4.9) and an AEAD (c.f. Definition 4.3 on Page 85). It also uses a so-called *key schedule* KS which we model as a keyed function $\mathsf{KS} : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^*$, where $\mathcal{K}$ matches the AKEM's key space. KS outputs an AEAD key $k$ and an initialisation vector nonce (called *base nonce* in the RFC) from which the AEAD's nonces are computed. (The key schedule defined in the HPKE standard also outputs an additional key called *exporter secret* that can be used to derive keys for use by arbitrary higher-level applications. This export API is not part of the single-shot encryption API that we are analysing, and thus we omit it in our definitions.) Listing 4.8 gives the formal specification of APKE built from AKEM, KS and AEAD.

We observe that in the single-shot encryption API, every AEAD key $k$ is used to produce exactly one ciphertext, and thus is only used with one nonce. In HPKE, messages are counted with a sequence number $s$ starting at 0 and the nonce for a message is computed by $\mathrm{nonce} \oplus s$. For the single-shot encryption API this means that the nonce is equal to the initialisation vector nonce. At the same time, this means that nonce is by definition unique.

We now give theorems stating the $(n,q_e,q_d,q_c)$-Outsider-CCA, $(n,q_e,q_d)$-Outsider-Auth and $(n,q_e,q_d,q_c)$-Insider-CCA security of APKE[AKEM, KS, AEAD] defined in Listing 4.8. Theorems 4.3 to 4.5 are proven using CryptoVerif version 2.05. This version includes an improvement in the computation of probability bounds that allows us to express these bounds as functions of the total numbers of queries to the AENC, ADEC, and CHALL

oracles instead of the number of users and the numbers of queries per user. The CryptoVerif input files are given in hpke.auth.outsider-cca.ocv, hpke.auth.insider-cca.ocv, and hpke.auth.outsider-auth.ocv [Alw+]. These proofs are fairly straightforward. As an example, we prefer explaining the proof of Theorem 4.7 later, which is more interesting. In Section 4.5.4, we show that APKE[AKEM, KS, AEAD] cannot achieve Insider-Auth security.

[Alw+] Alwen et al., *Analysing the HPKE Standard Supplementary Material*

As detailed in Section 4.3, we define a multi-key PRF security experiment $(n_k, q_{\mathsf{PRF}})$-PRF with $n_k$ keys, in which the adversary makes at most $q_{\mathsf{PRF}}$ queries for each key. We also define multi-key IND-CPA and INT-CTXT security experiments for the AEAD: $n_k$-IND-CPA and $(n_k, q_d)$-INT-CTXT, with $n_k$ keys, in which the adversary makes at most one encryption query for each key and, for the INT-CTXT experiment, at most $q_d$ decryption queries in total. In these experiments, the nonces of the AEAD are chosen randomly.

**Theorem 4.3** (AKEM Outsider-CCA+KS PRF+AEAD IND-CPA+AEAD INT-CTXT ⇒ APKE Outsider-CCA). *For any* $(n, q_e, q_d, q_c)$-Outsider-CCA *adversary* $\mathcal{A}$ *against* APKE[AKEM, KS, AEAD], *there exist an* $(n, q_e + q_c, q_d)$-Outsider-CCA *adversary* $\mathcal{B}$ *against* AKEM, *an* $(q_c, q_c + q_d)$-PRF *adversary* $\mathcal{C}$ *against* KS, *an* $q_c$-IND-CPA *adversary* $\mathcal{D}_1$ *against* AEAD *and an* $(q_c, q_d)$-INT-CTXT *adversary* $\mathcal{D}_2$ *against* AEAD *such that* $t_{\mathcal{B}} \approx t_{\mathcal{A}}$, $t_{\mathcal{C}} \approx t_{\mathcal{A}}$, $t_{\mathcal{D}_1} \approx t_{\mathcal{A}}$, $t_{\mathcal{D}_2} \approx t_{\mathcal{A}}$, *and*

$$\begin{aligned}\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Outsider-CCA}}_{\mathcal{A},\mathsf{APKE[AKEM,KS,AEAD]}} \leq\ & 2 \cdot \mathsf{Adv}^{(n,q_e+q_c,q_d)\text{-Outsider-CCA}}_{\mathcal{B},\mathsf{AKEM}} + 2 \cdot \mathsf{Adv}^{(q_c,q_c+q_d)\text{-PRF}}_{\mathcal{C},\mathsf{KS}} \\ & + 2 \cdot \mathsf{Adv}^{q_c\text{-IND-CPA}}_{\mathcal{D}_1,\mathsf{AEAD}} + 2 \cdot \mathsf{Adv}^{(q_c,q_d)\text{-INT-CTXT}}_{\mathcal{D}_2,\mathsf{AEAD}} \\ & + 6n^2 \cdot P_{\mathsf{AKEM}} \,.\end{aligned}$$

**Theorem 4.4** (AKEM Insider-CCA+KS PRF+AEAD IND-CPA+AEAD INT-CTXT ⇒ APKE Insider-CCA). *For any* $(n, q_e, q_d, q_c)$-Insider-CCA *adversary* $\mathcal{A}$ *against* APKE[AKEM, KS, AEAD], *there exist an* $(n, q_e, q_d, q_c)$-Insider-CCA *adversary* $\mathcal{B}$ *against* AKEM, *an* $(q_c, q_c + q_d)$-PRF *adversary* $\mathcal{C}$ *against* KS, *an* $q_c$-IND-CPA *adversary* $\mathcal{D}_1$ *against* AEAD *and an* $(q_c, q_d)$-INT-CTXT *adversary* $\mathcal{D}_2$ *against* AEAD *such that* $t_{\mathcal{B}} \approx t_{\mathcal{A}}$, $t_{\mathcal{C}} \approx t_{\mathcal{A}}$, $t_{\mathcal{D}_1} \approx t_{\mathcal{A}}$, $t_{\mathcal{D}_2} \approx t_{\mathcal{A}}$, *and*

$$\begin{aligned}\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Insider-CCA}}_{\mathcal{A},\mathsf{APKE[AKEM,KS,AEAD]}} \leq\ & 2 \cdot \mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Insider-CCA}}_{\mathcal{B},\mathsf{AKEM}} + 2 \cdot \mathsf{Adv}^{(q_c,q_c+q_d)\text{-PRF}}_{\mathcal{C},\mathsf{KS}} \\ & + 2 \cdot \mathsf{Adv}^{q_c\text{-IND-CPA}}_{\mathcal{D}_1,\mathsf{AEAD}} + 2 \cdot \mathsf{Adv}^{(q_c,q_d)\text{-INT-CTXT}}_{\mathcal{D}_2,\mathsf{AEAD}} \\ & + 6n^2 \cdot P_{\mathsf{AKEM}} \,.\end{aligned}$$

**Theorem 4.5** (AKEM Outsider-CCA+AKEM Outsider-Auth+KS PRF+AEAD INT-CTXT ⇒ APKE Outsider-Auth). *For any* $(n, q_e, q_d)$-Outsider-Auth *adversary* $\mathcal{A}$ *against* APKE[AKEM, KS, AEAD], *there exist an* $(n, q_e, q_d + 1)$-Outsider-CCA *adversary* $\mathcal{B}_1$ *against* AKEM, *an* $(n, q_e, q_d + 1)$-Outsider-Auth *adversary* $\mathcal{B}_2$ *against* AKEM, *an* $(q_e + q_d + 1, q_e + 2q_d + 1)$-PRF *adversary* $\mathcal{C}$ *against* KS, *and an* $(q_e + q_d + 1, 2q_d + 1)$-INT-CTXT *adversary* $\mathcal{D}$ *against* AEAD *such that* $t_{\mathcal{B}_1} \approx t_{\mathcal{A}}$, $t_{\mathcal{B}_2} \approx t_{\mathcal{A}}$, $t_{\mathcal{C}} \approx t_{\mathcal{A}}$, $t_{\mathcal{D}} \approx t_{\mathcal{A}}$, *and*

$$\begin{aligned}\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-Auth}}_{\mathcal{A},\mathsf{APKE[AKEM,KS,AEAD]}} \leq\ & \mathsf{Adv}^{(n,q_e,q_d+1)\text{-Outsider-CCA}}_{\mathcal{B}_1,\mathsf{AKEM}} + \mathsf{Adv}^{(n,q_e,q_d+1)\text{-Outsider-Auth}}_{\mathcal{B}_2,\mathsf{AKEM}} \\ & + \mathsf{Adv}^{(q_e+q_d+1,q_e+2q_d+1)\text{-PRF}}_{\mathcal{C},\mathsf{KS}} \\ & + \mathsf{Adv}^{(q_e+q_d+1,2q_d+1)\text{-INT-CTXT}}_{\mathcal{D},\mathsf{AEAD}} + n(q_e + 11n) \cdot P_{\mathsf{AKEM}} \,.\end{aligned}$$

### 4.5.4 *Infeasibility of* Insider-Auth *security*

For any AKEM, KS, and AEAD, the construction APKE[AKEM, KS, AEAD] given in Listing 4.8 is not $(n, q_e, q_d)$-Insider-Auth secure. The inherent reason for this construction to be vulnerable against this attack is that the KEM ciphertext does not depend on the message. Thus, the KEM ciphertext can be reused and the DEM ciphertext can be exchanged by the encryption of any other message.

**Theorem 4.6.** *There exists an efficient adversary $\mathcal{A}$ against* $(1,1,0)$-Insider-Auth *security of* APKE[AKEM, KS, AEAD] *such that*

$$\mathsf{Adv}^{(1,1,0)\text{-Insider-Auth}}_{\mathcal{A},\mathsf{APKE}[\mathsf{AKEM},\mathsf{KS},\mathsf{AEAD}]} = 1 \ ,$$

*which means that $\mathcal{A}$ makes a single query to* GEN*, a single query to* AENC *and no query to* ADEC*.*

*Proof.* We construct adversary $\mathcal{A}$ in Listing 4.9. It has oracle access to GEN, AENC and ADEC. It first creates a key pair using the GEN oracle to receive a public key $\mathrm{pk}_1$. It then generates a challenge key pair $(\mathrm{sk}^*, \mathrm{pk}^*)$ and queries the AENC oracle on any index 1, receiver public key $\mathrm{pk}^*$, an arbitrary message $m_1$, as well as arbitrary associated data $\mathrm{aad}$ and string $\mathrm{info}$.

Listing 4.9: Adversary $\mathcal{A}$ against $(1,1,0)$-Insider-Auth security (as defined in Listing 4.7) of APKE[AKEM, KS, AEAD].

Adversary $\mathcal{A}^{\text{GEN},\text{AENC},\text{ADEC}}$

```
(pk_1, ℓ = 1) ← GEN
(sk*, pk*) ← AKEM.Gen
m_1 := aad := info := 1
(c_1, c_2) ← AENC(1, pk*, m_1, aad, info)
K ← AuthDecap(sk*, pk_1, c_1)
(k, nonce) ← KS(K, info)
m_2 := 2
c'_2 ← AEAD.Enc(k, m_2, aad, nonce)
return (1, sk*, (c_1, c'_2), aad, info)
```

The challenger computes $(c_1, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}_1, \mathrm{pk}^*)$, $(k, \mathrm{nonce}) \leftarrow \mathsf{KS}(K, \mathrm{info})$ and $c_2 \leftarrow \mathsf{AEAD.Enc}(k, m_1, \mathrm{aad}, \mathrm{nonce})$, and returns $(c_1, c_2)$ to $\mathcal{A}$.

Since $\mathcal{A}$ knows the secret key $\mathrm{sk}^*$, it is able to compute the underlying KEM key $K$ using AuthDecap. Next, it computes $(k, \mathrm{nonce})$ and thus retrieves the key $k$ used in the AEAD scheme. Finally, $\mathcal{A}$ encrypts any other message $m_2$ to ciphertext $c'_2$ and replaces the AEAD ciphertext $c_2$ with the new ciphertext. Since $(c_1, c_2) \neq (c_1, c'_2)$, the latter constitutes a valid forgery in the $(1,1,0)$-Insider-Auth security experiment. □

## 4.6 THE HPKE STANDARD

In Section 4.6.1, we show how to construct HPKE's abstract AKEM construction DH-AKEM from a nominal group $\mathcal{N}$ and a key derivation function

Listing 4.10: DH-AKEM[$\mathcal{N}$, KDF] = (Gen, AuthEncap, AuthDecap) as defined in the RFC [Bar+22], constructed from a nominal group $\mathcal{N}$ and key derivation function KDF : $\{0,1\}^* \to \mathcal{K}$, with $\mathcal{K} = \{0,1\}^N$.

```
Gen
01 sk ←$ E_H
02 pk ← g^sk
03 return (sk, pk)

ExtractAndExpand(dh, context)
04 IKM ← "HPKE-v1" || suite_id ||
          "eae_prk" || dh
05 info ← Encode(N) || "HPKE-v1" ||
          suite_id || "shared_secret" ||
          context
06 return KDF(empty, IKM, info)

AuthEncap(sk ∈ E_H, pk ∈ G)
07 (esk, epk) ←$ Gen
08 context ← (epk, pk, g^sk)
09 dh ← (pk^esk, pk^sk)
10 K ← ExtractAndExpand(dh, context)
11 return (epk, K)

AuthDecap(sk ∈ E_H, pk ∈ G, epk ∈ G)
12 context ← (epk, g^sk, pk)
13 dh ← (epk^sk, pk^sk)
14 return ExtractAndExpand(dh, context)
```

KDF. In Section 4.6.2, we define and analyse HPKE's specific key schedule $\mathsf{KS}_{\mathsf{Auth}}$ and key derivation function $\mathsf{HKDF}_N$. Finally, in Section 4.6.3 we put everything together and obtain the HPKE standard in Auth mode from all previous sections.

### 4.6.1 *HPKE's AKEM Construction* DH-AKEM

In this section we present the RFC's instantiation of the AKEM definition, and prove that it satisfies the security notions defined earlier. Listing 4.10 shows the formal definition of DH-AKEM[$\mathcal{N}$, KDF] relative to a nominal group $\mathcal{N}$ (c.f. Definition 4.4) and a key derivation function KDF : $\{0,1\}^* \to \mathcal{K}$, where $\mathcal{K}$ is the key space. (The RFC uses a key space $\mathcal{K}$, consisting of bitstrings of length $N$, which corresponds to `Nsecret` in the RFC.) The construction also depends on the fixed-size protocol constants `"HPKE-v1"` and $suite_{id}$, where $suite_{id}$ identifies the KEM in use: it is a string `"KEM"` plus a two-byte identifier of the KEM algorithm. The bitstring Encode($N$) is the two-byte encoding of the length $N$ expressed in bytes. Correctness follows by property 1 of Definition 4.4. We make the implicit convention that AuthEncap and AuthDecap return reject ($\perp$) if their inputs are not of the right data type as specified in Listing 4.10.

We continue with statements about the $(n, q_e, q_d)$-Outsider-CCA, $(n, q_e, q_d, q_c)$-Insider-CCA, and $(n, q_e, q_d)$-Outsider-Auth security of DH-AKEM[$\mathcal{N}$, KDF], modelling KDF as a random oracle. The proofs are written with CryptoVerif version 2.05; the input files are dhkem.auth.outsider-cca-lr.ocv, dhkem.auth.insider-cca-lr.ocv, and dhkem.auth.outsider-auth-lr.ocv [Alw+]. We sketch the proof of one of the three theorems as an example, to help understand CryptoVerif's approach.

[Alw+] Alwen et al., *Analysing the HPKE Standard Supplementary Material*

Our results hold for any rerandomisable nominal group, which covers the three NIST curves allowed by the RFC, as well as for the other two allowed curves, Curve25519 and Curve448. The bounds given in Theorems 4.7 to 4.9 depend on the probabilities $\Delta_{\mathcal{N}}$ and $P_{\mathcal{N}}$, which can be instantiated for these five different curves using the values indicated in Table 4.2 on Page 109.

The RFC mandates that implementations abort if the Diffie-Hellman shared secret is the point at infinity for P-256, P-384, and P-521, or the all-zero value for Curve25519 and Curve448. The security notions in this work do not contain this check, and the theorems are valid for both implementa-

tions that do or do not contain it (this is because the properties we prove are not concerned about contributiveness). More formally, the adversary could check for all public keys leading to problematic Diffie-Hellman shared secrets before calling the oracles in our security games.

At the end of this section, we sketch the attack against the Insider-Auth security.

**Theorem 4.7** (Outsider-CCA security of DH-AKEM)**.** *Let $\mathcal{N}$ be a rerandomisable nominal group. Under the* GDH *assumption in $\mathcal{N}$ and modelling* KDF *as a random oracle,* DH-AKEM$[\mathcal{N}, \mathsf{KDF}]$ *is* Outsider-CCA *secure. In particular, for any adversary $\mathcal{A}$ against $(n, q_e, q_d)$-*Outsider-CCA *security of* DH-AKEM$[\mathcal{N}, \mathsf{KDF}]$ *that issues at most $q_h$ queries to the random oracle* KDF*, there exists an adversary $\mathcal{B}$ against* GDH *such that*

$$\begin{aligned}\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-CCA}}_{\mathcal{A},\text{DH-AKEM}[\mathcal{N},\mathsf{KDF}]} \leq \mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B},\mathcal{N}} + (n+q_e)\cdot\Delta_{\mathcal{N}} \\ + (q_e q_d + 2nq_e + 10q_e^2 + 11n^2)\cdot P_{\mathcal{N}}\end{aligned}$$

*$\mathcal{B}$ issues $5q_h$ queries to the* DH *oracle, and $t_{\mathcal{B}} \approx t_{\mathcal{A}}$.*

*Proof.* This proof is mechanized using the tool CryptoVerif. We give to the tool the assumptions that $\mathcal{N}$ is a nominal group that satisfies the GDH assumption, formalized by Definition 4.7 and Theorem 4.1, and that KDF is a random oracle. We also give the definition of DH-AKEM, and ask it to show that the games $(n, q_e, q_d)$-Outsider-CCA$_\ell$ and $(n, q_e, q_d)$-Outsider-CCA$_r$ are computationally indistinguishable. In the particular case of DH-AKEM, these two games include an additional oracle: the random oracle KDF. The theorem, the initial game definitions, and the proof indications are available in the file dhkem.auth.outsider-cca-lr.ocv [Alw+].

[Alw+] Alwen et al., *Analysing the HPKE Standard Supplementary Material*

The proof proceeds by transforming the game $(n, q_e, q_d)$-Outsider-CCA$_\ell$ by several steps into a game $G_{\text{final}}$ and the game $(n, q_e, q_d)$-Outsider-CCA$_r$ into the same game $G_{\text{final}}$. Since all transformation steps performed by CryptoVerif are designed to preserve computational indistinguishability, we obtain that $(n, q_e, q_d)$-Outsider-CCA$_\ell$ and $(n, q_e, q_d)$-Outsider-CCA$_r$ are computationally indistinguishable. We guide the transformations with the following main steps.

Starting from $(n, q_e, q_d)$-Outsider-CCA$_\ell$, in the oracle AEncap, we first distinguish whether the provided public key $\mathrm{pk}$ is honest, by testing whether $\mathrm{pk} = \mathrm{pk}_i$ for some $i$ (a test that appears in $(n, q_e, q_d)$-Outsider-CCA$_r$). We rename some variables to give them different names when $\mathrm{pk} \in \{\mathrm{pk}_1, \ldots, \mathrm{pk}_n\}$ and when $\mathrm{pk} \notin \{\mathrm{pk}_1, \ldots, \mathrm{pk}_n\}$, to facilitate future game transformations. In the oracle ADecap, we test whether $\exists K : (\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{E}$, which corresponds to a test done in $(n, q_e, q_d)$-Outsider-CCA$_r$. Furthermore, when this test succeeds, we replace the result normally returned by ADecap, AuthDecap$(\mathrm{sk}_j, \mathrm{pk}, c)$ with the key $K$ found in $\mathcal{E}$. CryptoVerif shows that this replacement does not modify the result, which corresponds to the correctness of DH-AKEM. In the random oracle, we distinguish whether the argument received from the adversary has a format that matches the one used by DH-AKEM or not. Only when the format matches, this argument may coincide with a call to the hash oracle made from DH-AKEM. Next, we apply the random oracle assumption. Each call to the random oracle is

replaced with the following test: if the argument is equal to the argument of a previous call, we return the previous result; otherwise, we return a fresh random value. Finally, we apply the GDH assumption, which allows us to show that some comparisons between Diffie-Hellman values are false. In particular, CryptoVerif shows that the arguments of calls to the random oracle coming from $\mathrm{AEncap}$ with $\mathrm{pk} \in \{\mathrm{pk}_1, \ldots, \mathrm{pk}_n\}$ cannot coincide with arguments of other calls. Hence, they return a fresh random key, as in $(n, q_e, q_d)$-Outsider-CCA$_r$.

Starting from $(n, q_e, q_d)$-Outsider-CCA$_r$, in the random oracle, we distinguish whether the argument received from the adversary has a format that matches the one used by DH-AKEM or not. Next, we apply the random oracle assumption, as we did on the left-hand side.

The transformed games obtained respectively from $(n, q_e, q_d)$-Outsider-CCA$_\ell$ and from $(n, q_e, q_d)$-Outsider-CCA$_r$ are then equal, which concludes the proof.

CryptoVerif computes the bound on the probability of distinguishing the games $(n, q_e, q_d)$-Outsider-CCA$_\ell$ and $(n, q_e, q_d)$-Outsider-CCA$_r$ by adding bounds computed at each transformation step. During this proof, CryptoVerif automatically eliminates unlikely collisions, in particular between public Diffie-Hellman keys. By default, CryptoVerif eliminates these collisions aggressively, even when that is not required for the proof to succeed, which results in a large probability bound. To avoid that, we allow the tool to eliminate collisions of probability $P_{\mathcal{N}}$ times a power 4 in $n$, $q_e^{\text{per user}}$, and $q_d^{\text{per user}}$, where $q_e^{\text{per user}}$ and $q_d^{\text{per user}}$ are the number of $\mathrm{AEncap}$ and $\mathrm{ADecap}$ queries respectively, per user. But we do not allow eliminating collisions with more than a power 4 in $n$, $q_e^{\text{per user}}$, and $q_d^{\text{per user}}$, nor collisions that involve $q_h$. □

**Theorem 4.8** (Insider-CCA security of DH-AKEM)**.** *Let $\mathcal{N}$ be a rerandomisable nominal group. Under the* GDH *assumption in $\mathcal{N}$ and modelling* KDF *as a random oracle,* DH-AKEM$[\mathcal{N}, \mathsf{KDF}]$ *is* Insider-CCA *secure. In particular, for any $(n, q_e, q_d, q_c)$-*Insider-CCA *adversary $\mathcal{A}$ against* DH-AKEM$_{\mathcal{N}}$ *that issues at most $q_h$ queries to the random oracle, there exists an adversary $\mathcal{B}$ against* GDH *such that*

$$\begin{aligned}\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Insider-CCA}}_{\mathcal{A},\text{DH-AKEM}[\mathcal{N},\mathsf{KDF}]} &\leq \mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B},\mathcal{N}} + (n+q_c)\cdot\Delta_{\mathcal{N}} \\ &\quad + (q_c q_d + 4q_c q_e + 2nq_e + 10q_e^2 + 3q_c^2 + 9n^2)\cdot P_{\mathcal{N}}\end{aligned}$$

*$\mathcal{B}$ makes $5q_h$ queries to the* DH *oracle, and $t_{\mathcal{B}} \approx t_{\mathcal{A}}$.*

**Theorem 4.9** (Outsider-Auth security of DH-AKEM)**.** *Let $\mathcal{N}$ be a rerandomisable nominal group. Under the* sqGDH *assumption in $\mathcal{N}$ and modelling* KDF *as a random oracle,* DH-AKEM$[\mathcal{N}, \mathsf{KDF}]$ *is* Outsider-Auth *secure. In particular, for any $(n, q_e, q_d)$-*Outsider-Auth *adversary $\mathcal{A}$ against* DH-AKEM$_{\mathcal{N}}$ *that issues at most $q_h$ queries to the random oracle, there exists an adversary $\mathcal{B}$ against* sqGDH *such that*

$$\begin{aligned}\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-Auth}}_{\mathcal{A},\text{DH-AKEM}[\mathcal{N},\mathsf{KDF}]} &\leq 2\mathsf{Adv}^{\mathsf{sqGDH}}_{\mathcal{B},\mathcal{N}} + 2(n+q_e)\cdot\Delta_{\mathcal{N}} \\ &\quad + (2q_e q_d + 4nq_d + 16q_e^2 + 4nq_e + 12n^2)\cdot P_{\mathcal{N}}\end{aligned}$$

*$\mathcal{B}$ issues $7q_h$ queries to the* DH *oracle, and $t_{\mathcal{B}} \approx t_{\mathcal{A}}$.*

INFEASIBILITY OF Insider-Auth SECURITY. As for APKE, we could define an Insider-Auth security notion for AKEM, which precludes forgeries even when the receiver key pair is dishonest, provided the sender key pair is honest. However, the DH-AKEM construction does not even achieve KCI security, a relaxation of Insider-Auth security only precluding forgeries for leaked, but still honestly generated, receiver key pairs. Indeed, in DH-AKEM, knowledge of an arbitrary receiver secret key is already sufficient to compute the Diffie-Hellman shared key for any sender public key. Thus, in a KCI attack, an adversary that learns a target receiver's keys can trivially produce a KEM ciphertext and corresponding encapsulated key for any target sender public key.

### 4.6.2 *HPKE's Key Schedule and Key Derivation Function*

HPKE's key schedule $\mathsf{KS}_{\mathsf{Auth}}$ and key derivation function $\mathsf{HKDF}_N$ are both instantiated via the functions Extract and Expand which are defined below. We proceed to prove a theorem that $\mathsf{KS}_{\mathsf{Auth}}$ is a PRF, as needed for the composition results presented in Theorems 4.3 to 4.5. Then, we argue why $\mathsf{HKDF}_N$ can be modelled as a random oracle, as assumed by Theorems 4.7 to 4.9 on DH-AKEM. Finally, we indicate how the entire $\mathsf{HPKE}_{\mathsf{Auth}}$ scheme is assembled from the individual building blocks presented in the previous sections.

Extract AND Expand. The RFC defines two functions Extract and Expand as follows.

- $\mathsf{Extract}(\mathrm{salt}, \mathrm{IKM})$ is a function keyed by a bitstring $\mathrm{salt}$, with input keying material $\mathrm{IKM}$ as parameter, and returns a bitstring of fixed length $N_h$ bits.
- $\mathsf{Expand}(\mathrm{PRK}, \mathrm{info}, L)$ is a function keyed by $\mathrm{PRK}$, with an arbitrary bitstring $\mathrm{info}$ and a length $L$ as parameters, and returns a bitstring of length $L$.

In Theorem 4.10, we assume that Extract and Expand are PRFs with the first parameter being the PRF key. HPKE instantiates Extract and Expand with HMAC-SHA-2, for which the PRF assumption is justified by [Bel15; BCK96]. (Generally, HPKE's instantiation of Expand uses HMAC iteratively to achieve the variable output length $L$. However, all values $L$ used in HPKE are less or equal than the output length of one HMAC call.) We also assume that Extract is collision resistant, provided its keys are not larger than blocks of SHA-2, which is needed to avoid that the keys be hashed before computing HMAC, and true in HPKE. This property is immediate from the collision resistance of SHA-2, studied in [GH04].

[Bel15] Bellare, "New Proofs for NMAC and HMAC: Security without Collision Resistance"

[BCK96] Bellare et al., "Keying Hash Functions for Message Authentication"

[GH04] Gilbert and Handschuh, "Security Analysis of SHA-256 and Sisters"

KEY SCHEDULE. The key schedule $\mathsf{KS}_{\mathsf{Auth}}$ serves as a bridging step between the AKEM and the AEAD of APKE. The computations done by $\mathsf{KS}_{\mathsf{Auth}}$ are as indicated in Listing 4.11. The function KeySchedule used internally is the common key schedule function that the RFC defines for all modes. In $\mathsf{HPKE}_{\mathsf{Auth}}$, the $\mathrm{mode}$ parameter is set to the constant one-byte value `0x02` identifying the mode Auth. Similarly, mode Auth does not use a pre-shared key, so the $\mathrm{psk}$ parameter is always set to the empty string $\mathrm{empty}$, and the value $\mathrm{psk\_id}$ that is identifying which pre-shared key is used, is equally set

Listing 4.11: The key schedule $\mathsf{KS}_{\mathsf{Auth}}$ used in $\mathsf{HPKE}_{\mathsf{Auth}}$ [Bar+22].

```
KS_Auth(k_PRF, info)
01 return KeySchedule(k_PRF, 0x02, info, empty, empty)

KeySchedule(k_PRF, mode, info, psk, psk_id)
02 context ← mode ||
               LabeledExtract(empty, "psk_id_hash", psk_id) ||
               LabeledExtract(empty, "info_hash"  , info)
03 secret ← LabeledExtract(k_PRF, "secret", psk)
04 k ← LabeledExpand(secret, "key", context, N_k)
05 nonce ← LabeledExpand(secret, "base_nonce", context, N_n)
06 return (k, nonce)

LabeledExtract(salt, label, IKM')
07 return Extract(salt, "HPKE-v1" || suite_id || label || IKM')

LabeledExpand(PRK, label, context, L)
08 return Expand(PRK, Encode(L) || "HPKE-v1" || suite_id || label || context, L)
```

to empty. The RFC defines LabeledExtract and LabeledExpand as wrappers around Extract and Expand, for domain separation and context binding. The value $suite_{id}$ is a 10-byte string identifying the ciphersuite, composed as a concatenation of the string "HPKE", and two-byte identifiers of the KEM, the KDF, and the AEAD algorithm in use. The bitstring Encode($L$) is the two-byte encoding of the length $L$ expressed in bytes. The values $N_k$ and $N_n$ indicate the length of the AEAD key and nonce.

The composition results established by Theorems 4.3 to 4.5 assume that $\mathsf{KS}_{\mathsf{Auth}}$ is a PRF. The following theorem proves this property for $\mathsf{HPKE}_{\mathsf{Auth}}$'s instantiation of $\mathsf{KS}_{\mathsf{Auth}}$.

**Theorem 4.10** (Extract CR + Extract PRF + Expand PRF ⇒ $\mathsf{KS}_{\mathsf{Auth}}$ PRF)**.** *Assuming that* Extract *is a collision-resistant hash function for calls with the labels* "psk_id_hash" *and* "info_hash"*, that* Extract *is a* PRF *for calls with the label* "secret"*, and that* Expand *is a* PRF*, it follows that* $\mathsf{KS}_{\mathsf{Auth}}$ *is a* PRF.

*In particular, for any* $(n_k, q_{\mathsf{PRF}})$-PRF *adversary* $\mathcal{A}$ *against* $\mathsf{KS}_{\mathsf{Auth}}$*, there exist an adversary* $\mathcal{B}$ *against the collision resistance of* Extract*, a* $(n_k, n_k)$-PRF *adversary* $\mathcal{C}_1$ *against* Extract*, and a* $(n_k, 2q_{\mathsf{PRF}})$-PRF *adversary* $\mathcal{C}_2$ *against* Expand *such that* $t_{\mathcal{B}} \approx t_{\mathcal{A}}$*,* $t_{\mathcal{C}_1} \approx t_{\mathcal{A}}$*,* $t_{\mathcal{C}_2} \approx t_{\mathcal{A}}$*, and*

$$\mathsf{Adv}^{(n_k,q_{\mathsf{PRF}})\text{-PRF}}_{\mathcal{A},\mathsf{KS}_{\mathsf{Auth}}} \leq \mathsf{Adv}^{\mathsf{CR}}_{\mathcal{B},\mathsf{Extract}} + \mathsf{Adv}^{(n_k,n_k)\text{-PRF}}_{\mathcal{C}_1,\mathsf{Extract}} + \mathsf{Adv}^{(n_k,2q_{\mathsf{PRF}})\text{-PRF}}_{\mathcal{C}_2,\mathsf{Expand}} .$$

This theorem is proven by CryptoVerif in keyschedule.auth.prf.ocv [Alw+].

[Alw+] Alwen et al., *Analysing the HPKE Standard Supplementary Material*

THE KEY DERIVATION FUNCTION KDF IN DH-AKEM. The AKEM instantiation DH-AKEM as we defined it in Listing 4.10 uses a function KDF to derive the KEM shared secret. In $\mathsf{HPKE}_{\mathsf{Auth}}$, this function is instantiated by $\mathsf{HKDF}_N$, as defined in Listing 4.12, using the above-defined Extract and Expand internally. The output length $N$ corresponds to Nsecret in the RFC.

Listing 4.12: Function $\mathsf{HKDF}_N[\mathsf{Extract}, \mathsf{Expand}]$ as used in $\mathsf{HPKE}_{\mathsf{Auth}}$.

$\mathsf{HKDF}_N(\mathrm{salt}, \mathrm{IKM}, \mathrm{info})$
01 $\mathrm{PRK} \leftarrow \mathsf{Extract}(\mathrm{salt}, \mathrm{IKM})$
02 **return** $\mathsf{Expand}(\mathrm{PRK}, \mathrm{info}, N)$

In the analysis of the key schedule presented above, we assume that Extract and Expand are pseudo-random functions. However, this assumption would not be sufficient to prove the security of DH-AKEM: the random oracle model is required. The simplest choice is to assume that the whole key derivation function $\mathsf{KDF} = \mathsf{HKDF}_N$ is a random oracle, as we do in Theorems 4.7 to 4.9. (Alternatively, we could probably rely on some variant of the PRF-ODH assumption [Bre+17]. While in principle the PRF-ODH assumption is weaker than the random oracle model, Brendel, Fischlin, Günther, and Janson [Bre+17] show that it is implausible to instantiate the PRF-ODH assumption without a random oracle, so that would not make a major difference.) The invocations of Extract and Expand in DH-AKEM and $\mathsf{KS}_{\mathsf{Auth}}$ use different labels for domain separation, so choosing different assumptions is sound. Next, we further justify the random oracle assumption for $\mathsf{HKDF}_N$.

[Bre+17] Brendel et al., "PRF-ODH: Relations, Instantiations, and Impossibility Results"

As mentioned at the beginning of Section 4.6, HPKE instantiates Extract and Expand with HMAC [KBC97], which makes $\mathsf{HKDF}_N$ exactly the widely-used HKDF key derivation function [KE10]. HPKE specifies SHA-2 as the hash function underlying HMAC. Lemma 3.9 in Chapter 3 shows that HKDF is indifferentiable from a random oracle under the following assumptions[3]: (1) HMAC is indifferentiable from a random oracle. For HMAC-SHA-2, this is justified by Theorem 4.4 in the full version of [Dod+12] assuming the compression function underlying SHA-2 is a random oracle. The theorem's restriction on HMAC's key size is fulfilled, because DH-AKEM uses either the empty string, or a bitstring of hash output length as key. (2) Values of $\mathrm{IKM}$ do not collide with values of $\mathrm{info} \,||\, \texttt{0x01}$. This is guaranteed by the prefix `"HPKE-v1"` of $\mathrm{IKM}$, which is used as a prefix for $\mathrm{info}$ as well, but shifted by two characters, because the two-byte encoding of the length $N$ comes before it. The shared secret lengths `Nsecret` specified in the RFC correspond exactly to the output length of the hash function; this means there is only one internal call to Expand, and thus we do not need to consider collisions of $\mathrm{IKM}$ with the input to later HMAC calls.

[KBC97] Krawczyk et al., *HMAC: Keyed-Hashing for Message Authentication*

[KE10] Krawczyk and Eronen, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*

[3] The exact probability bound is indicated in Lemma 8 of that paper's full version.

[Dod+12] Dodis et al., "To Hash or Not to Hash Again? (In)Differentiability Results for $H^2$ and HMAC"

### 4.6.3 *HPKE's APKE Scheme* $\mathsf{HPKE}_{\mathsf{Auth}}$

Let $\mathsf{HPKE}_{\mathsf{Auth}} := \mathsf{APKE}[\mathsf{DH\text{-}AKEM}[\mathcal{N}, \mathsf{HKDF}_N], \mathsf{KS}_{\mathsf{Auth}}, \mathsf{AEAD}]$ be the construction of APKE obtained by applying the black-box AKEM/DEM composition of Listing 4.8 to the $\mathsf{DH\text{-}AKEM}[\mathcal{N}, \mathsf{HKDF}_N]$ authenticated KEM (Listing 4.10), where $\mathcal{N}$ is a rerandomisable nominal group. For the key schedule of $\mathsf{HPKE}_{\mathsf{Auth}}$ we use $\mathsf{KS}_{\mathsf{Auth}}$ of Listing 4.11 and for the key derivation function we use $\mathsf{HKDF}_N$ of Listing 4.12. For both $\mathsf{KS}_{\mathsf{Auth}}$ and $\mathsf{HKDF}_N$ we implement the Extract and Expand functions using HMAC (as described in the HPKE specification). Finally, we instantiate HMAC using one of the SHA2 family of hash functions. (Which one depends on the target bit security

of $\mathsf{HPKE}_{\mathsf{Auth}}$, as we discuss below.)

The AKEM/DEM composition Theorems 4.3 to 4.5, together with Theorem 4.10 on the key schedule $\mathsf{KS}_{\mathsf{Auth}}$, and Theorems 4.7 to 4.9 on DH-AKEM's security, and $P_{\mathsf{DH\text{-}AKEM}} = P_{\mathcal{N}}$ provide the following concrete security bounds for $\mathsf{HPKE}_{\mathsf{Auth}}$. For simplicity, we ignore all constants and set $q := q_e + q_d + q_c$.

$$\begin{aligned}
\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Outsider-CCA}}_{\mathcal{A},\mathsf{HPKE}_{\mathsf{Auth}}} &\leq \mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B}_1,\mathcal{N}} + (n+q)^2 \cdot P_{\mathcal{N}} + (n+q) \cdot \Delta_{\mathcal{N}} \\
&\quad + \mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C},\mathsf{KS}_{\mathsf{Auth}}} + \mathsf{Adv}^{q\text{-IND-CPA}}_{\mathcal{D}_1,\mathsf{AEAD}} + \mathsf{Adv}^{(q,q)\text{-INT-CTXT}}_{\mathcal{D}_2,\mathsf{AEAD}} \\
\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-Auth}}_{\mathcal{A},\mathsf{HPKE}_{\mathsf{Auth}}} &\leq \mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B}_1,\mathcal{N}} + \mathsf{Adv}^{\mathsf{sqGDH}}_{\mathcal{B}_2,\mathcal{N}} + (n+q)^2 \cdot P_{\mathcal{N}} + (n+q) \cdot \Delta_{\mathcal{N}} \\
&\quad + \mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C},\mathsf{KS}_{\mathsf{Auth}}} + \mathsf{Adv}^{(q,q)\text{-INT-CTXT}}_{\mathcal{D}_1,\mathsf{AEAD}} .
\end{aligned}$$

The bound for Insider-CCA is the same as the one for Outsider-CCA. In all bounds, we have $\mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C},\mathsf{KS}_{\mathsf{Auth}}} \leq \mathsf{Adv}^{\mathsf{CR}}_{\mathcal{C}_1,\mathsf{Extract}} + \mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C}_2,\mathsf{Extract}} + \mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C}_3,\mathsf{Expand}}$. Moreover, the adversaries $\mathcal{B}_1, \mathcal{B}_2, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ have (roughly) the same running time as $\mathcal{A}$.

PARAMETER CHOICES OF $\mathsf{HPKE}_{\mathsf{Auth}}$. The HPKE standard allows different choices of rerandomisable nominal groups $\mathcal{N}$ to obtain a concrete instance of $\mathsf{HPKE}_{\mathsf{Auth}}$, that lead to different bounds on the statistical parameters $P_{\mathcal{N}}$ and $\Delta_{\mathcal{N}}$. The standard also fixes the length $N$ of the KEM keyspace, c.f. Table 4.2. Even though lengths are expressed in bytes in the RFC and the implementation, we express them in bits in this section as this is more convenient to discuss the number of bits of security.

All concrete instances of $\mathsf{HPKE}_{\mathsf{Auth}}$ proposed by the HPKE standard build Extract and Expand from HMAC which, in turn, uses a hash function. HPKE proposes several concrete hash functions (all in the SHA2 family). For our security bounds, the relevant consequence of choosing a particular hash function is the resulting key length $N_h$ of Expand when used as a PRF, c.f. Table 4.3.

Finally, to instantiate $\mathsf{HPKE}_{\mathsf{Auth}}$, we must also specify the AEAD scheme. HPKE allows for several choices which affect the AEAD key length $N_k$, nonces length $N_n$, and tag length $N_t$, c.f. Table 4.4.

DISCUSSION. We say that an instance of $\mathsf{HPKE}_{\mathsf{Auth}}$ achieves $\kappa$ *bits of security* if the success ratio $\mathsf{Adv}_{\mathcal{A},\mathsf{HPKE}_{\mathsf{Auth}}}/t_{\mathcal{A}}$ is upper bounded by $2^{-\kappa}$ for any adversary $\mathcal{A}$ with runtime $t_{\mathcal{A}} \leq 2^{\kappa}$. In particular, we say that a term $\varepsilon$ *has* $\kappa$ *bits of security* if $\varepsilon/t_{\mathcal{A}} \leq 2^{-\kappa}$. We discuss the implications of our results for the bit security of the various instances of $\mathsf{HPKE}_{\mathsf{Auth}}$ proposed by the standard.

The runtime $t_{\mathcal{A}}$ of any adversary $\mathcal{A}$ in an APKE security game is lower-bounded by $n+q$, since the adversary needs $n$ steps to parse the $n$ public keys and additional $q$ steps to make the oracle queries. We assume that $t_{\mathcal{A}} \leq 2^{\kappa}$, where $\kappa$ is the target security level.

We now estimate the security level supported by each term in $\mathsf{Adv}_{\mathcal{A},\mathsf{HPKE}_{\mathsf{Auth}}}$.

- **Term** $\mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B}_1,\mathcal{N}}$. Rerandomisable nominal groups $\mathcal{N}$ proposed for use by the HPKE standard were designed to provide $\kappa_{\mathcal{N}}$ bits of security (c.f. Table 4.2). That is, we assume that $\mathsf{Adv}^{\mathsf{GDH}}_{\mathcal{B}_1,\mathcal{N}}/t_{\mathcal{B}_1} \leq 2^{-\kappa_{\mathcal{N}}}$. Since $t_{\mathcal{A}} \approx t_{\mathcal{B}_1}$, we conclude that this term has $\kappa_{\mathcal{N}}$ bits of security. The same arguments hold for $\mathsf{Adv}^{\mathsf{sqGDH}}_{\mathcal{B}_2,\mathcal{N}}$.
- **Term** $(n+q)^2 \cdot P_{\mathcal{N}}$. Let us show that this term also has $\kappa_{\mathcal{N}}$ bits of security. We have $n+q \leq t_{\mathcal{A}}$. Thus, it suffices to show that $(n+q) \cdot P_{\mathcal{N}} \leq 2^{-\kappa_{\mathcal{N}}}$.

TABLE 4.2: Parameters of DH-AKEM[$\mathcal{N}$, HKDF$_N$] depending on the choice of the rerandomisable nominal group $\mathcal{N}$.

| | **P-256** | **P-384** | **P-521** | **Curve25519** | **Curve448** |
|---|---|---|---|---|---|
| Security level $\kappa_{\mathcal{N}}$ (bits) | 128 | 192 | 256 | 128 | 224 |
| $P_{\mathcal{N}} \leq$ | $2^{-255}$ | $2^{-383}$ | $2^{-520}$ | $2^{-250}$ | $2^{-444}$ |
| $\Delta_{\mathcal{N}} \leq$ | 0 | 0 | 0 | $2^{-126}$ | $2^{-221}$ |
| KEM keyspace $N$ (bits) | 256 | 384 | 512 | 256 | 512 |

TABLE 4.3: Choices of HMAC and the PRF key lengths of Expand, instantiated with HMAC.

| | **HMAC-SHA256** | **HMAC-SHA384** | **HMAC-SHA512** |
|---|---|---|---|
| PRF key length $N_h$ of Expand (bits) | 256 | 384 | 512 |

TABLE 4.4: Choices of the AEAD scheme and their parameters.

| | **AES-128-GCM** | **AES-256-GCM** | **ChaCha20-Poly1305** |
|---|---|---|---|
| AEAD key length $N_k$ (bits) | 128 | 256 | 256 |
| AEAD nonces length $N_n$ (bits) | 96 | 96 | 96 |
| AEAD tag length $N_t$ (bits) | 128 | 128 | 128 |

Since $t_{\mathcal{A}} \leq 2^{\kappa_{\mathcal{N}}}$, we get that $(n+q) \leq 2^{\kappa_{\mathcal{N}}}$. The statement now follows as, according to Table 4.2, $P_{\mathcal{N}} \lesssim 2^{-2\kappa_{\mathcal{N}}}$.

- **Term** $(n+q) \cdot \Delta_{\mathcal{N}}$. Let us show that this term also has $\kappa_{\mathcal{N}}$ bits of security. For all NIST curves, we have $\Delta_{\mathcal{N}} = 0$ trivially implying the statement. In contrast, for Curve25519 and Curve448, $\Delta_{\mathcal{N}} \lesssim 2^{-\kappa_{\mathcal{N}}}$, so $(n+q) \cdot \Delta_{\mathcal{N}} \approx (n+q) \cdot 2^{-\kappa_{\mathcal{N}}}$. As $n+q \leq t_{\mathcal{A}}$, the statement also holds for these curves.
- **Term** $\mathsf{Adv}^{\mathsf{CR}}_{\mathcal{C}_1,\mathsf{Extract}}$. The output length $N_h$ of the concrete hash functions are listed in Table 4.3. Since the generic bound on collision resistance is $t^2_{\mathcal{C}_1}/2^{N_h}$, this term has $N_h/2$ bits of security.
- **Term** $\mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C}_3,\mathsf{Expand}}$. The PRF key lengths $N_h$ of Expand are specified in Table 4.3. Modelling the PRF as a random oracle, we have $\mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C}_3,\mathsf{Expand}} \leq q^2/2^{N_h}$. So this term also has $N_h/2$ bits of security.
- **Term** $\mathsf{Adv}^{(q,q)\text{-PRF}}_{\mathcal{C}_2,\mathsf{Extract}}$. The PRF key length $N$ of Extract is specified in Table 4.2. By the same argument as for the previous term, this term has $N/2$ bits of security. Since $N/2 \geq \kappa_{\mathcal{N}}$ by Table 4.2, this term has $\kappa_{\mathcal{N}}$ bits of security.
- **Terms** $\mathsf{Adv}^{q\text{-IND-CPA}}_{\mathcal{D}_1,\mathsf{AEAD}} + \mathsf{Adv}^{(q,q)\text{-INT-CTXT}}_{\mathcal{D}_2,\mathsf{AEAD}}$. The terms refer to the multi-key security of the AEAD schemes (c.f. Section 4.3), studied for instance in [BT16]. However, the current results are not sufficient to guarantee the expected security level, such as 128 bits for AES-128-GCM. We recommend further research to study the exact bounds of the terms instantiated with the AEAD schemes from Table 4.4. In any case a simple key/nonce-collision attack has success probability

[BT16] Bellare and Tackmann, "The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3"

$\mathsf{Adv}^{q\text{-IND-CPA}}_{\mathcal{D}_1,\mathsf{AEAD}} = q^2/2^{N_k+N_n}$, where $N_k$ is the AEAD key length and $N_n$ is the nonce length. A simple computation shows that this term has at most $N_k$ bits of security (assuming $q \leq 2^{N_n}$). Moreover, a simple attack against INT-CTXT by guessing the authentication tag has success probability $\mathsf{Adv}^{(q,q)\text{-INT-CTXT}}_{\mathcal{D}_2,\mathsf{AEAD}} = q/2^{N_t}$, where $N_t$ is the length of the authentication tag. Hence, this term has at most $N_t$ bits of security. Assuming these attacks also serve as an upper bound, these terms would have $\min(N_k, N_t)$ bits of security if $q \leq 2^{N_n}$. Since for all AEAD schemes of Table 4.4, we have $N_t = 128$ bits, that limits the security level of HPKE to 128 bits.

To sum up, the analysis above suggests that HPKE has about $\kappa = \min(\kappa_{\mathcal{N}}, N_h/2, N_k, N_t)$ bits of security, under the assumption that $t_{\mathcal{A}} \leq 2^{\kappa}$ and $q \leq 2^{N_n}$. Since the tag length of the AEAD is $N_t = 128$ bits, we obtain $\kappa = 128$ bits; a greater security level could be obtained by using AEADs with longer tags. More research on the multi-key security of AEAD schemes is still needed to confirm this analysis.

ACKNOWLEDGEMENTS. The authors would like to thank the HPKE RFC co-authors Richard Barnes, Karthikeyan Bhargavan, and Christopher Wood for fruitful discussions during the preparation of this work. The authors also thank Pierre Boutry and Christian Doczkal for very helpful feedback on a previous version of this work.

Bruno Blanchet was supported by ANR TECAP (decision number ANR-17-CE39-0004-03). Eduard Hauck was supported by the DFG SPP 1736 Big Data. Eike Kiltz was supported by the BMBF iBlockchain project, the EU H2020 PROMETHEUS project 780701, the DFG SPP 1736 Big Data, and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. Benjamin Lipp was supported by ERC CIRCUS (grant agreement no. 683032) and ANR TECAP (decision number ANR-17-CE39-0004-03). Doreen Riepel was funded by the DFG under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

# 5

# *The Hybrid Public Key Encryption Standard*

This chapter gives an overview of our other involvements in analysis, development, and implementation of the HPKE standard. Section 5.1 presents a preliminary analysis done in CryptoVerif before the detailed work on $\mathsf{HPKE}_{\mathsf{Auth}}$, presented in Chapter 4, was started. Section 5.2.5 describes an implementation of HPKE in the F* programming language. Section 5.2 gives insights about how the two analyses and the implementation influenced the HPKE standard.

## 5.1 An Analysis of All HPKE Modes

This section is based on the preprint "An Analysis of Hybrid Public Key Encryption" [Lip20] that presents a preliminary analysis of all modes of HPKE using CryptoVerif. More precisely, it proves asymptotic cryptographic guarantees for the single-shot encryption and the secret export interfaces of all modes. The analysis looks at the HPKE construction as a whole, without decomposing it into its KEM and DEM building blocks like the analysis presented in Chapter 4. It was started at a time when the HPKE standard was still at an early stage, at Draft 2 of overall 12 drafts that it went through. For this reason, we start by laying out how the cryptographic dependencies and algorithm definitions differ from the version analysed in Chapter 4. From a cryptographic point-of-view, the version analyzed in Chapter 4 is the final version. Changes in later drafts did not concern the protocol.

[Lip20] Lipp, *An Analysis of Hybrid Public Key Encryption*

### 5.1.1 *Hybrid Public Key Encryption at Draft 2*

In Draft 2, the KEM and the KeySchedule function were defined differently than in the final version. The most significant changes were made to them in Draft 3 as a result of feedback we provided to the authors of the standard. This is described in more detail in Section 5.2.

Key Encapsulation Mechanism (KEM). Similarly to how the Authenticated KEM is defined in Definition 4.9, HPKE Draft 2 uses a standard KEM for $\mathsf{HPKE}_{\mathsf{Base}}$ and $\mathsf{HPKE}_{\mathsf{PSK}}$. A KEM consists of a tuple of algorithms $(\mathsf{Gen}, \mathsf{Encap}, \mathsf{Decap})$:

- $\mathsf{Gen}$ is the same as in Definition 4.9.

- Encap takes as input a receiver public key $\mathrm{pk}$, and outputs an encapsulation $c$ and a shared secret $K \in \mathcal{K}$.
- Deterministic Decap takes as input a receiver secret key $\mathrm{sk}$ and an encapsulation $c$, and outputs a shared key $K \in \mathcal{K}$.

Similarly to the authenticated variant, we require that for all $(\mathrm{sk}, \mathrm{pk}) \in \mathsf{Gen}$

$$\Pr_{(c,K)\xleftarrow{\$}\mathsf{Encap}(\mathrm{pk})}[\mathsf{Decap}(\mathrm{sk}, c) = K] = 1 \ .$$

The sets of secret and public keys $\mathcal{SK}$ and $\mathcal{PK}$, the projection function $\mu$, and the key collision probability are the same as for the authenticated variant, as both use the same key generation algorithm.

In this analysis, we focus on the instantiation by the Diffie-Hellman-based KEM defined in the HPKE standard. We recall how Encap and AuthEncap were defined at the time of Draft 2:

- Encap($\mathrm{pkR}$): Given input $\mathrm{pkR}$, output a ciphertext $\mathrm{enc}$ of size $N_{\mathrm{enc}}$ bytes and shared key $\mathrm{zz}$ of size $N_{\mathrm{pk}}$ bytes:

$$\begin{aligned}(\mathrm{skE}, \mathrm{pkE}) &\leftarrow_{\$} \mathsf{Gen}() \\ \mathrm{zz} &\leftarrow \mathrm{pkR}^{\mathrm{skE}} \\ \mathrm{enc} &\leftarrow \mathrm{pkE}\end{aligned}$$

- AuthEncap($\mathrm{pkR}, \mathrm{skS}$): Given input $\mathrm{pkR}$ and $\mathrm{skS}$, output a ciphertext $\mathrm{enc}$ of size $N_{\mathrm{enc}}$ bytes and shared key $\mathrm{zz}$ of size $2N_{\mathrm{pk}}$ bytes:

$$\begin{aligned}(\mathrm{skE}, \mathrm{pkE}) &\leftarrow_{\$} \mathsf{Gen}() \\ \mathrm{zz} &\leftarrow \mathrm{pkR}^{\mathrm{skE}} \,||\, \mathrm{pkR}^{\mathrm{skS}} \\ \mathrm{enc} &\leftarrow \mathrm{pkE}\end{aligned}$$

The most notable difference to the final version is that the Diffie-Hellman shared secret is directly used as KEM shared secret, and not first processed by a key derivation function. The consequences of this are discussed in Section 5.2.

HASH FUNCTION AND KEY DERIVATION FUNCTION (KDF). HPKE Draft 2 defined a KDF as a tuple of deterministic algorithms (Hash, Extract, Expand) used for secret derivation and expansion. The functions Extract and Expand are defined as in Section 4.6.2. Hash($m$) is defined as a function taking a bitstring $m$ as parameter and returning its hash of length $N_h$ bits. The difference of Draft 2 compared to the final version is how these functions are used. The final version does not use any Hash function directly, and it uses Extract and Expand only via labeled wrappers. We will see this in the definition of the KeySchedule function, shortly.

KEY SCHEDULE. The KeySchedule function is defined as follows, where $\mathrm{zz}$ and $\mathrm{enc}$ are the shared secret and the ciphertext returned by the KEM, and $\mathrm{pkR}$ and $\mathrm{pkS}$ are the receiver and sender public keys. The other parameters are as in the final version.

KeySchedule($\mathrm{zz}, \mathrm{enc}, \mathrm{mode}, \mathrm{pkR}, \mathrm{info}, \mathrm{psk}, \mathrm{psk\_id}, \mathrm{pkS}$) :

$$
\begin{aligned}
\mathrm{ciphersuite} &\leftarrow \mathrm{kem\_id} \,||\, \mathrm{kdf\_id} \,||\, \mathrm{aead\_id} \\
\mathrm{psk\_id\_hash} &\leftarrow \mathsf{Hash}(\mathrm{psk\_id}) \\
\mathrm{info\_hash} &\leftarrow \mathsf{Hash}(\mathrm{info}) \\
\mathrm{ctx} &\leftarrow \mathrm{mode} \,||\, \mathrm{ciphersuite} \,||\, \mathrm{enc} \,||\, \mathrm{pkR} \,||\, \mathrm{pkS} \,||\, \mathrm{psk\_id\_hash} \,||\, \mathrm{info\_hash} \\
s &\leftarrow \mathsf{Extract}(\mathrm{psk}, \mathrm{zz}) \\
k &\leftarrow \mathsf{Expand}(s, \text{“hpke key”} \,||\, \mathrm{ctx}, N_k) \\
n &\leftarrow \mathsf{Expand}(s, \text{“hpke nonce”} \,||\, \mathrm{ctx}, N_n) \\
s_{\mathrm{exp}} &\leftarrow \mathsf{Expand}(s, \text{“hpke exp”} \,||\, \mathrm{ctx}, N_h)
\end{aligned}
$$

In the Base and Auth modes, $\mathrm{psk}$ and $\mathrm{psk\_id}$ are set to all-zero default values. In the Base and PSK modes, $\mathrm{pkS}$ is set to an all-zero default value. The values $\mathrm{kem\_id}$, $\mathrm{kdf\_id}$, and $\mathrm{aead\_id}$ are 2-byte identifiers of the algorithm in use, respectively. The key schedule function returns an *encryption context* containing the symmetric key $k$, nonce $n$, and the exporter secret $s_{\mathrm{exp}}$. This encryption context can then be used to call the encryption and secret export interfaces.

The differences to the final version of the KeySchedule are: the hash function is used directly on $\mathrm{psk\_id}$ and $\mathrm{info}$; the context bitstring includes $\mathrm{enc}$, $\mathrm{pkR}$, and $\mathrm{pkS}$, which have been moved into the context of the KEM key derivation in the final version; Extract and Expand are used instead of LabeledExtract and LabeledExpand.

AEAD. Authenticated Encryption with Additional Data is defined as in Definition 4.3, there is no difference to the final version.

APPLICATION INTERFACE. The encryption function exposed by the single-shot encryption interface is defined as follows:

Context.Seal($\mathrm{aad}, \mathrm{pt}$):

$$\mathrm{ct} \leftarrow \mathsf{AEAD.Enc}(k, \mathrm{pt}, \mathrm{aad}, n)$$

Here, $\mathrm{pt}$ is the plaintext, and $\mathrm{aad}$ the additional data that is authenticated by the AEAD. The interface uses the values from the encryption context.

The secret export interface takes as input a context string $\mathrm{ctx}$ and a desired length $L$ in bytes and produces a secret derived from the internal exporter secret using the Expand function of the KDF:

Context.Export($\mathrm{ctx}, L$):

$$k_{\mathrm{exp}} \leftarrow \mathsf{Expand}(s_{\mathrm{exp}}, \text{“exp key”} \,||\, \mathrm{ctx}, L)$$

### 5.1.2 *Analysis*

STRUCTURE OF THE SECURITY GAME. In this analysis, we prove a two-user security notion for HPKE. In the setup of the game, we create key pairs $(\mathrm{skS}, \mathrm{pkS}), (\mathrm{skR}, \mathrm{pkR})$ and a pre-shared key $\mathrm{psk}$ for two honest users $S$ and

$R$, and sample a secret bit $b$. Then, we give the adversary access to an encryption and a decryption oracle that can be called $q_e$ and $q_d$ times.

The encryption oracle acts as user $S$ and takes a receiver public key, two plaintexts $p_0, p_1$ of same length, and additional data $\mathrm{aad}$ as parameter. Internally, the oracle prepares an HPKE encryption context for the appropriate mode, using the receiver public key provided by the adversary. It always uses the honest sender private key and pre-shared key, in the relevant modes, respectively. Then, it encrypts $p_b$ using the encryption interface and exports two secrets using the secret export interface, with two different but constant contexts. Finally, the oracle returns the KEM and the AEAD ciphertext and the two exported secrets.

The decryption oracle acts as user $R$ and takes a KEM and an AEAD ciphertext and additional data $\mathrm{aad}$ as parameter and returns whether the decryption was successful. Internally, it prepares an HPKE encryption context using the honest receiver $R$'s private key, honest sender $S$'s public key, and their pre-shared key, depending on the mode.

Depending on the mode, we give the adversary access to oracles that leak the private keys and the pre-shared key of $S$ and $R$. We do not allow compromise scenarios that would trivially break secrecy. In $\mathsf{HPKE}_{\mathsf{Base}}$, we do not allow any compromise. In $\mathsf{HPKE}_{\mathsf{PSK}}$, we allow compromise of either $\mathrm{skR}$ or $\mathrm{psk}$. In $\mathsf{HPKE}_{\mathsf{Auth}}$, we allow compromise of $\mathrm{skS}$. In $\mathsf{HPKE}_{\mathsf{AuthPSK}}$, we allow combinations of compromising $\mathrm{skS}$, $\mathrm{skR}$, and the $\mathrm{psk}$, but disallow scenarios where the $\mathrm{psk}$ and $\mathrm{skR}$ are both compromised.

PROOF GOALS. We attempt to prove two secrecy properties for sessions between the honest users $S$ and $R$: secrecy of the bit $b$ and real-or-random indistinguishability of the exported secrets in all modes. For $\mathsf{HPKE}_{\mathsf{PSK}}$, $\mathsf{HPKE}_{\mathsf{Auth}}$, and $\mathsf{HPKE}_{\mathsf{AuthPSK}}$, we attempt to prove sender authentication in scenarios where it is not trivially broken: in $\mathsf{HPKE}_{\mathsf{PSK}}$ for sessions before compromise of the $\mathrm{psk}$; in $\mathsf{HPKE}_{\mathsf{Auth}}$ for sessions before compromise of $\mathrm{skS}$; in $\mathsf{HPKE}_{\mathsf{AuthPSK}}$ for sessions before the compromise of the $\mathrm{psk}$, and for sessions before the compromise of $\mathrm{skS}$ and $\mathrm{skR}$ (because HPKE is vulnerable to key-compromise impersonation).

We use the following CryptoVerif queries to express the secrecy proof goals:

```
query secret b.
query secret export1_secr public_vars export2_secr.
query secret export2_secr public_vars export1_secr.
```

Here, we rely on CryptoVerif's built-in support for real-or-random indistinguishability. Using the `public_vars` keyword, we prove secrecy of an export key even if the other one was compromised.

We use events and a correspondance query to express the authentication properties for $\mathsf{HPKE}_{\mathsf{PSK}}$, $\mathsf{HPKE}_{\mathsf{Auth}}$, $\mathsf{HPKE}_{\mathsf{AuthPSK}}$. In the model, we issue two events `sent` and `rcvd` just before the sender sends a message and just after the recipient successfully decrypts a message. We ask CryptoVerif to prove the following correspondence:

$$\begin{aligned} &\mathsf{event}(\mathsf{rcvd}(\mathrm{true}, \mathrm{mode}, \mathrm{pkR}, \mathrm{pkS}, \mathrm{psk\_id}, \mathrm{info}, \mathrm{aad}, \mathrm{pt}, k_{\mathrm{exp},1}, k_{\mathrm{exp},2}) \\ \Rightarrow\ &\mathsf{event}(\mathsf{sent}(\quad \mathrm{mode}, \mathrm{pkR}, \mathrm{pkS}, \mathrm{psk\_id}, \mathrm{info}, \mathrm{aad}, \mathrm{pt}, k_{\mathrm{exp},1}, k_{\mathrm{exp},2}) \end{aligned}$$

We only prove a non-injective correspondence, which means that we do not prove that *each* rcvd event has a *unique* corresponding sent event. Instead, a single sent event can satisfy the condition for arbitrary many matching rcvd events. This is because HPKE does not provide protection against replay. The last two parameters are only used in the export variants of the model and absent in the oneshot variants. The first parameter is $\mathrm{true}$ only in scenarios where sender authentication is possible; in others we do not attempt the proof.

CRYPTOGRAPHIC ASSUMPTIONS. We use the following cryptographic assumptions. We assume that Extract is indifferentiable from a random oracle. We assume that Expand is a PRF, where the first parameter is the key. We write proofs for two different assumptions on Hash: that it is a collision-resistant hash function, and that it is indifferentiable from a random oracle. We assume that the AEAD scheme used is IND-CPA- and INT-CTXT-secure [BN00]. We assume that the KEM uses a prime-order group that satisfies the Gap Diffie-Hellman (GDH) assumption [OP01]. We also assume that an implementation of HPKE validates public keys before usage.

[BN00] Bellare and Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm"

[OP01] Okamoto and Pointcheval, "The Gap-Problems: a New Class of Problems for the Security of Cryptographic Schemes"

Due to the specific length parameters used with Extract and Expand, both are implemented by exactly one call to hmac. Thus, we are using a PRF and a random oracle assumption on the same building block. By the following reasoning, we establish that the hmac call inside Extract operates on a different input domain than the ones inside Expand: The first argument to Extract's hmac call is the $\mathrm{psk}$ which has length $N_h$. The first argument to all hmac calls inside Expand is the result of Extract; and this has length $N_h$ as well. Thus, the first argument does not directly separate the input domains. The second argument to Extract's hmac call is the result from the DH operation. This has either length $N_{\mathrm{pk}}$ or $2N_{\mathrm{pk}}$. The second argument to all hmac calls inside Expand has *at least* the length of the $\mathrm{ctx}$ variable, which is defined as:

$$\begin{aligned}\mathrm{ciphersuite} = \mathsf{concat}(&\overbrace{\mathrm{kem\_id}}^{2\,\mathrm{bytes}}, \overbrace{\mathrm{kdf\_id}}^{2\,\mathrm{bytes}}, \overbrace{\mathrm{aead\_id}}^{2\,\mathrm{bytes}})\\ \mathrm{ctx} = \mathsf{concat}(&\underbrace{\mathrm{mode}}_{1\,\mathrm{byte}}, \underbrace{\mathrm{ciphersuite}}_{6\,\mathrm{bytes}}, \underbrace{\mathrm{enc}}_{N_{\mathrm{enc}}}, \underbrace{\mathrm{pkRm}}_{N_{\mathrm{pk}}}, \underbrace{\mathrm{pkSm}}_{N_{\mathrm{pk}}},\\ &\underbrace{\mathrm{pskID\_hash}}_{N_h}, \underbrace{\mathrm{info\_hash}}_{N_h})\end{aligned} \tag{5.1}$$

With this, we conclude that for all modes of HPKE, the length of $\mathrm{ctx}$ is strictly greater than the length of $\mathrm{zz}$. In turn, the input domain of Extract's hmac call is different from the input domains to the hmac calls inside Expand.

RESULTS. Using CryptoVerif's interactive mode, we succeed to write proofs for the desired properties in all modes and compromise scenarios using the above cryptographic assumptions. [1] The only exception are the proofs for the export variant in mode AuthPSK for a collision-resistant hash function, which we were not able to complete at the time.

[1] The model files of this analysis can be accessed at https://github.com/blipp/hpke-analysis-material.

DISCUSSION AND LIMITATIONS. This analysis consistutes a first confirmation of asymptotic security of the HPKE standard's desired security properties. However, it comes with several cryptographic limitations. The proofs are only

valid for prime-order Diffie-Hellman groups, although HPKE is also specified for Curve25519 and Curve448. For $\mathsf{HPKE}_{\mathsf{Auth}}$, we provide an analysis in Chapter 4 that also covers these elliptic curves using the framework of nominal groups. An analysis of the other modes in this framework is left for future work. In the authenticated modes, the model does not currently allow that the sender encrypts a message to its own public key. Furthermore, the adversary could get more power to strengthen the security notion: it could be allowed to choose the user's private keys, and not only to compromise them (this has been implemented in the analysis presented in Chapter 4); it could be allowed to choose the context used for secret export. Generally, the analysis of the so-called *multi-shot* interface is left for future work, where multiple messages can be encrypted and multiple secrets can be exported. Finally, a modular analysis of HPKE using the KEM/DEM paradigm would be desirable for all modes. For $\mathsf{HPKE}_{\mathsf{Auth}}$, this has been done as presented in Chapter 4.

## 5.2 Influence of Our Analyses on the Standard

Before and after I became an official co-author of the HPKE standard, I contributed a total of 39 pull requests that got merged. [2] In the following, we cover those that are related to the cryptographic design of HPKE.

[2] https://github.com/cfrg/draft-irtf-cfrg-hpke/pulls?q=is%3Apr+author%3Ablipp

### 5.2.1 *Making the KEM IND-CCA-Secure*

After the analysis presented in Section 5.1, we started a modular analysis of HPKE using the KEM/DEM paradigm. As a start, we tried to prove that the variant of DHKEM used in $\mathsf{HPKE}_{\mathsf{Base}}$ and $\mathsf{HPKE}_{\mathsf{PSK}}$ is IND-CCA-secure. However, it turned out that DHKEM of HPKE Draft 2 was not provably IND-CCA-secure on its own, using the Gap Diffie-Hellman assumption. We use this assumption because an HPKE recipient can be used as an oracle for the Decisional Diffie-Hellman problem (see Definition 4.5). We recall that at that time, as presented in Section 5.1.1, DHKEM returned the Diffie-Hellman shared secret directly as KEM shared secret. Trying to prove that $\mathrm{pkR}^{\mathrm{skE}}$ is indistinguishable from a random group element $Z$ fails, because the adversary can submit $\mathrm{pkE}$, $\mathrm{pkR}$, and $\mathrm{pkR}^{\mathrm{skE}}$ (or $Z$) to the DDH oracle. Thus, the Diffie-Hellman shared secret cannot be used directly as shared secret of the KEM. The usual mitigation is to use a key derivation function to compute the KEM shared secret.

Then, a proof using the random oracle assumption can be sketched as follows. Let $H$ be a random oracle, and assume the KEM shared secret is computed by $\mathrm{zz} \leftarrow H(\mathrm{pkR}^{\mathrm{skE}})$. Assume that the adversary attempts to distinguish $\mathrm{zz}$ from a random value by querying the random oracle with values called $Z$. If $Z = \mathrm{pkR}^{\mathrm{skE}}$, the random oracle returns $\mathrm{zz}$. By the GDH assumption, the probability that the adversary computes $\mathrm{pkR}^{\mathrm{skE}}$ is negligible, and so we can replace the boolean $Z = \mathrm{pkR}^{\mathrm{skE}}$ by `false` in a game transformation. In the new game, the random oracle never returns $\mathrm{zz}$ to the adversary, and thus, the adversary cannot distinguish $\mathrm{zz}$ from a random value.

With this in mind, it becomes clear that the security proof for HPKE as a whole, as described in Section 5.1, succeeded because we assumed Extract in the KeySchedule to be a random oracle.

We proposed a change to the HPKE standard introducing a key derivation function call within DHKEM. [3] It became part of HPKE as of Draft 3.

[3] "Modifications for IND-CCA-secure DHKEM and independent random oracles" https://github.com/cfrg/draft-irtf-cfrg-hpke/pull/50

Another reason to use a key derivation function to derive the final KEM shared secret is that practical pseudo-random functions require a key that is a uniformly random *bitstring*, and not only a uniformly random group element. In general, the bitstring representation of Diffie-Hellman public keys is not a surjective function, and so, the bitstring representation of uniformly random group elements could be distinguished from uniformly random bitstrings.

### 5.2.2 *Identity Binding*

As presented in Section 5.1.1, the KEM ciphertext and the public keys of sender and receiver were used as context of the key derivation inside KeySchedule. In the pull request mentioned above, we moved these three values into the context of the KEM's key derivation to strengthen its IND-CCA security against *identity mis-binding issues*, also known as *unkown key-share attacks* (see Section 3.2.4 and Section 3.6 for our informal and formal definitions used when analysing WireGuard). X25519 and X448 are particularly prone to this because they have "equivalent" public keys, as we discussed already in Chapter 3: These are public keys that are different, but lead to the *same* Diffie-Hellman shared secret when used with the same private key. Putting the bitstring representation of the public keys explicitly into the context of the key derivation makes sure that the KEM shared secret ends up different, even for equivalent public keys.

Generally, as mentioned in the discussion of the WireGuard analysis in Section 3.7, including as much identity information as possible in the context of a key derivation increases security and simplifies the formal analysis.

### 5.2.3 *Oracle Separation by Domain Separation*

In our WireGuard analysis, we gave a detailed reasoning why using a random-oracle and a collision-resistance assumption for the same hash function is sound, by carefully analysing the input domains of its different usages (Section 3.4.5). When introducing the key derivation function into DHKEM, we wanted to make analysis as easy as possible with regards to the use of hash functions, as in practice, it is possible that the same hash function is used within DHKEM and for the remainder of HPKE. As a start, we decided to not use the hash function directly, but only via Extract and Expand, to avoid the need to consider how Extract and Expand use the hash functions internally. Then, we gave each usage a unique label as prefix, such that the set of labels is prefix-free (i. e., no label is the prefix of another label). The result of this design is what we presented in Section 4.6.1 for DHKEM and Listing 4.11 for KeySchedule. The term "Oracle Separation" stems from the discussion after the presentation of the paper [BDG20] at Eurocrypt 2020, where Daniel J. Bernstein suggested this term. [4]

[BDG20] Bellare et al., "Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability"

[4] https://chat.iacr.org/#narrow/stream/47-RWC-2021/topic/TUE.3A.20Post-quantum.20Cryptography/near/18995

### 5.2.4 *Influences of the Analysis of* $\mathsf{HPKE}_{\mathsf{Auth}}$

When we started working on the analysis of $\mathsf{HPKE}_{\mathsf{Auth}}$, the KeySchedule function of HPKE was defined slightly differently: Line 03 of Listing 4.11 was not

$$s \leftarrow \mathsf{LabeledExtract}(k_{\mathsf{PRF}}, \texttt{"secret"}, \mathrm{psk})$$

but

$$\mathrm{psk\_hash} \leftarrow \mathsf{LabeledExtract}(\mathrm{empty}, \texttt{"psk\_hash"}, \mathrm{psk})$$
$$s \leftarrow \mathsf{LabeledExtract}(\mathrm{psk\_hash}, \texttt{"secret"}, k_{\mathsf{PRF}}).$$

We suggested this change, and it was integrated into Draft 6, in time before we submitted the paper. [5] Swapping the KEM shared secret $k_{\mathsf{PRF}}$ and the psk allows us to use a standard PRF assumption on HKDF-Extract, using the KEM shared secret as uniformly random PRF key. In the previous version, this was problematic: in general, the psk cannot be used as PRF key because it is not present in all modes, and in particular not in $\mathsf{HPKE}_{\mathsf{Auth}}$; using the second parameter of Extract as PRF key is not straightforward because there, the KEM shared secret $k_{\mathsf{PRF}}$ is prefixed by several constants, making it non-uniformly random (see Line 07 in Listing 4.11). While this change improves the situation for proofs based on the security of the KEM shared secret $k_{\mathsf{PRF}}$, the question of how to prove security of $\mathsf{HPKE}_{\mathsf{PSK}}$ and $\mathsf{HPKE}_{\mathsf{AuthPSK}}$ based on the psk in case of compromised $k_{\mathsf{PRF}}$ remains for future work. The RFC does not require the psk to be drawn from a uniform distribution, so a PRF assumption with the psk as PRF key would not cover all cases, even if applicable in a straightforward way. The only requirement that the RFC imposes on the psk is that it must have at least "32 bytes of entropy" [Bar+22], following an analysis by Len, Grubbs, and Ristenpart, "effectively barring human-chosen passwords" [LGR21].

[5] "Use shared_secret as salt and psk as ikm in LabeledExtract" https://github.com/cfrg/draft-irtf-cfrg-hpke/pull/162

[Bar+22] Barnes et al., *Hybrid Public Key Encryption*

[LGR21] Len et al., "Partitioning Oracle Attacks"

Finally, the results of the analysis of $\mathsf{HPKE}_{\mathsf{Auth}}$ have been summarized in Section 9.1 "Security Properties" of the standard, notably to add text on the generic key-compromise impersonation attack that was surfaced by our work. Also, the composition theorems proved during the analysis have informed Section 9.2 "Security Requirements on a KEM used within HPKE" of the standard, which is important for when HPKE is used with other (Authenticated) KEMs.

### 5.2.5 *A Verified Implementation of HPKE and Input Length Limits*

A verified implementation of HPKE in the F* programming language has been developed and contributed to the HACL* library as prior work by other authors [Pol+20]. F* is a general-purpose functional programming language aimed at program verification [Swa+16]. HACL* is a cryptographic library written and verified in F* [Bha+17]. The HPKE implementation of [Pol+20] includes a high-level specification, and efficient low-level implementations that are optimized for multiple processor architectures. The specification and the implementations have been proved to be functionally equivalent. They are based on Draft 2 of HPKE, and only support the Base mode.

[Pol+20] Polubelova et al., "HACLxN: Verified Generic SIMD Crypto (for All Your Favourite Platforms)"

[Swa+16] Swamy et al., "Dependent Types and Multi-Monadic Effects in F*"

[Bha+17] Bhargavan et al., "HACL*: A Verified Modern Cryptographic Library"

As part of our involvement in the development of the HPKE standard, we updated the existing F* specification to the final version of HPKE, added

the remaining modes, added the secret export interface that was introduced with Draft 3, and added the DeriveKeyPair function that was introduced in Draft 5. This function allows to deterministically derive a KEM key pair from a bitstring value. We made sure the updated specification passes the test vectors provided by the RFC. For this, we extracted it to OCaml using F*'s built-in toolchain. We contributed this work to the HACL* library. [6] We leave the update of the efficient low-level implementations for future work.

[6] https://github.com/project-everest/hacl-star/tree/_blipp_hpke

As described in Section 4.6.2, HPKE uses HKDF to implement Extract and Expand. The specifications of HKDF-Extract and HKDF-Expand in HACL* indicate maximum input lengths for IKM and info that are enforced by the type system. These maximum lengths come from the specifications of the underlying hash functions. We propagate these maximum lengths from HKDF-Extract and HKDF-Expand through LabeledExtract and LabeledExpand all the way up to the input parameters of HPKE that were a priori not length-restricted by HPKE. At each step, we take into account the constant prefixes that are added to the function parameters. This way, we compute the maximum length of HPKE input parameters. The goal is to indicate them in the RFC as guidance for implementations. The affected input parameters are: psk, psk_id, and info as parameters of KeySchedule, ctx as parameter of the secret export interface, and the IKM input parameter of the DeriveKeyPair function. In the F* specification, we define the types of these input parameters taking into account their maximum lengths, parametrized by the hash function in use. [7] This specification is accepted by the F* type checker, which means that the computed maximum lengths are indeed valid input lengths, and in particular result in inputs to HKDF-Extract and HPKE-Expand with valid input lengths.

[7] We do this using refinement types, a concept that is explained later, in Section 6.6.1.

The concrete maximum lengths evaluated for all three hash functions specified for use with HPKE have been documented in Section 7.2.1 "Input Length Restrictions" in the RFC. [8] To give an example, the maximum length of the secret export's context parameter, when used with the SHA256 hash function, is $2^{61} - 120$ bytes. The maximum input length of the SHA256 hash function is $2^{61} - 1$ bytes, and thus, the maximum length of the context parameter is only marginally smaller. The difference is due to the HKDF construction and the constant prefixes used in HPKE. For practical usages, the HPKE standard recommends a limit of 64 bytes for the 5 input parameters mentioned above.

[8] Pull request for the change "Add section on input limits ..." https://github.com/cfrg/draft-irtf-cfrg-hpke/pull/117

# Part III

# LINKING COMPUTATIONAL PROOFS AND IMPLEMENTATIONS

# 6

## *cv2fstar: A Compiler from CryptoVerif to F⋆*

In the previous chapters of this thesis, we presented multiple case studies for mechanized cryptographic proofs done with CryptoVerif. Furthermore, we briefly mentioned the topic of verified implementations in F⋆, while writing about our implementation of HPKE in Section 5.2.5. The two worlds of CryptoVerif and F⋆ have so far not been connected in an automated way. In this chapter, we present cv2fstar, a compiler extending CryptoVerif. It compiles CryptoVerif models to executable F⋆ specifications that are set up to use the HACL⋆ library as cryptographic backend. With this, we close the remaining gap between CryptoVerif proofs and F⋆ implementations: cv2fstar produces F⋆ code that corresponds to the CryptoVerif model, and that keeps its cryptographic guarantees. The compiler translates CryptoVerif processes to pure F⋆ functions in state-passing style. Equations and some of the kinds of assumptions used by CryptoVerif are translated to F⋆ lemmas, as proof obligations for the implementation.

We showcase cv2fstar on the example of the Needham-Schroeder-Lowe (NSL) protocol. In its CryptoVerif model, we assume message encoding and decoding functions that are correct and that produce disjoint encodings for the three NSL protocol messages. Furthermore, serialization and deserialization functions for the types used in the protocol are assumed to exist and to be inverses of each other. cv2fstar translates these assumptions to F⋆ lemma statements that we prove for the specific functions chosen in the implementation. We implement the cryptographic building blocks used in the CryptoVerif model with already existing HACL⋆ functions. With this, the F⋆ specification generated by cv2fstar successfully executes as OCaml code in a toy example.

cv2fstar connects CryptoVerif to the large F⋆ ecosystem, eventually allowing to guarantee cryptographic properties on verified efficient low-level code – for example C code extracted with KaRaMeL.

In Section 6.1, we introduce and motivate cv2fstar in more detail. Section 6.2 gives an overview over related work. In Section 6.3, we briefly introduce the target language of our compiler. In Section 6.4, we present how the CryptoVerif input language is modified for cv2fstar. Section 6.5 gives an overview of the cv2fstar framework and the code generation of the cv2fstar compiler. Section 6.6 and Section 6.7 describe how types, and functions and constants in a CryptoVerif model are translated to F⋆. In Section 6.8, we discuss the polymorphic state type used in the F⋆ framework. Sections 6.9,

6.10, and 6.11 show the translation of tables, events, and terms. Section 6.12 explains how oracles are translated as F* functions. In Section 6.13, we describe the translation of equations and assumptions to lemmas. Section 6.14 clarifies the differences between cv2ocaml and cv2fstar. In Section 6.15, we showcase cv2fstar on the example of the Needham-Schroeder-Lowe protocol. Section 6.16 discusses the contributions of the work presented in this chapter. Finally, we discuss future work, both immediate and long-term, in Section 6.17.

The source code of cv2fstar consists of the extension to CryptoVerif, and the F* framework within which the generated F* code can be used. It is made available alongside the code for the NSL case study. [1]

[1] https://www.benjaminlipp.de/phd-thesis/

## 6.1 Motivation and Introduction

cv2fstar's goal is to establish a formal link between two analysis tools that work in different domains: CryptoVerif and F*.[2] Cryptographic security proofs done in CryptoVerif [Bla08] form an important part of past and current security research, as shown in Chapters 3 to 5 of this thesis about WireGuard and HPKE, and in analyses of TLS 1.3 [BBK17], Signal [KBB17], and SSH [CB13].

F* [Swa+16] is a "general-purpose functional programming language aimed at program verification".[3] F* has a rich type system, including dependent types and refinement types. Proofs can be done interactively and by discharging proof obligations to an SMT backend. F* programs can be extracted to OCaml and F# for execution, as well as to C when using the Low* subset of F* and the KaRaMeL[4] extraction tool [Pro+17], to WebAssembly [Pro+19], and to assembly code using Vale [Fro+19]. The F* ecosystem has been used to verify functional correctness and security of real-world protocols and systems, some of which we cite in the next section about related work. One important development that we want to mention here is the High-Assurance Cryptographic Library HACL* [Bha+17; Pol+20] that is written in F*. It has made its way into various production systems: Mozilla's NSS library, the Windows kernel, the Linux kernel, Microsoft's implementation of the QUIC protocol, the Tezos blockchain, and WireGuard.[5]

The motivation for a link between CryptoVerif and F* is twofold. First, we would like to have verified implementations that have concrete cryptographic security guarantees, and in particular by a mechanized formal link. Before cv2fstar, it was possible to write a CryptoVerif model and an F* specification such that their styles are sufficiently close to enable manual comparison and thus a more-or-less handwavy argument saying that they match. cv2fstar provides an automated translation between the two languages and allows protocols modeled in CryptoVerif to be embedded within F* programs. Together with the remainder of the F* ecosystem, we can then go all the way down to extracted C code that will have concrete cryptographic security guarantees. So, we preserve the cryptographic guarantees proved by CryptoVerif on the F* side, and can augment it with F*'s capabilities of proving functional correctness, memory safety, arithmetic safety, and secret-independent computation (e. g., against some timing side

[2] With my bachelor degree in physics in mind, I like to think about it as a great unification of forces, but it's the unification of verification forces – in this case CryptoVerif and F*.

[Bla08] Blanchet, "A Computationally Sound Mechanized Prover for Security Protocols"

[BBK17] Bhargavan et al., "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate"

[KBB17] Kobeissi et al., "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach"

[CB13] Cadé and Blanchet, "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH"

[Swa+16] Swamy et al., "Dependent Types and Multi-Monadic Effects in F*"

[3] https://www.fstar-lang.org/

[4] https://github.com/FStarLang/karamel

[Pro+17] Protzenko et al., "Verified Low-Level Programming Embedded in F*"

[Pro+19] Protzenko et al., "Formally Verified Cryptographic Web Applications in WebAssembly"

[Fro+19] Fromherz et al., "A Verified, Efficient Embedding of a Verifiable Assembly Language"

[Bha+17] Bhargavan et al., "HACL*: A Verified Modern Cryptographic Library"

[Pol+20] Polubelova et al., "HACLxN: Verified Generic SIMD Crypto (for All Your Favourite Platforms)"

[5] https://hacl-star.github.io/

channels). This is in particular interesting for non-cryptographic assumptions that the CryptoVerif model contains about functions, e. g., message encoding functions and (de)serialization functions that are assumed to be correctly encoding and decoding, and encryption and decryption functions that are assumed to be correctly implemented. In F*, we can implement these functions and prove that they actually have the correctness property assumed in the CryptoVerif model. The cv2fstar compiler translates the equations and non-cryptographic assumptions included in the CryptoVerif model to proof obligations – lemmas – in F*, which can then be proved to hold for the actual implementation. This is the first motivation for the link between CryptoVerif and F*, and it has been put in place for this first iteration of the cv2fstar project, and has been tested on the NSL case study.

Second, we would like to express the cryptographic theorems proved by CryptoVerif in F*, as assumed lemmas, or pre- and post-conditions of functions. Then, we could reason further with them, i. e., use them to prove other theorems. This can be interesting for larger systems that cannot be treated in CryptoVerif but where F*'s capabilities could be useful. This second motivation has not yet been implemented and remains for future work. The kinds of theorems that we need to translate are (1) correspondence properties between events, (2) secrecy properties, and (3) equivalence properties. While the translation of correspondence properties should be relatively straightforward with lemmas about lists of events, for secrecy and equivalence properties we need to relate program codes which is not straightforward in F*.

The cv2fstar project is joint work together with my PhD advisors Bruno Blanchet and Karthik Bhargavan. There is no publication or ongoing submission of this work yet, at the moment of thesis submission – this is planned for after submission when we did some finishing touches. Some of them are laid out in the section on future work. Future readers might want to watch out for a peer-reviewed publication of this work, or a preprint version.

## 6.2 Related Work

This work integrates in a long line of research considering different approaches to ensure cryptographic guarantees of implementations. We present work in the computational model and in the symbolic model. For both, we group by (1) extraction of an implementation from a model, (2) extraction of a model from an implementation, and (3) proofs done directly on an implementation.

Computational Model, Extracting Implementations. A translation from CryptoVerif models to OCaml has been developed during David Cadé's thesis [CB13; CB15]. It is accompagnied by a soundness proof, assuring that for each cryptographic attack successfully executed against the OCaml code, there exists an attack against the CryptoVerif model. A detailed comparison between cv2ocaml and cv2fstar is provided in Section 6.14. CertiCrypt [BGZB09] is built on top of Coq and can generate OCaml, Haskell, and Scheme implementations of its deeply embedded pWhile language through Coq's built-in extraction mechanism. The Foundational Cryptogra-

[CB13] Cadé and Blanchet, "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH"

[CB15] Cadé and Blanchet, "Proved Generation of Implementations from Computationally Secure Protocol Specifications"

[BGZB09] Barthe et al., "Formal Certification of Code-Based Cryptographic Proofs"

phy Framework (FCF) [PM15] allows to write security proofs with concrete bounds in the computational model. It is formalized in Coq, which again allows to extract executable code.

EXTRACTING MODELS. The Jasmin [Alm+17] framework and programming language together with the EasyCrypt proof assistant [Bar+11] form an ecosystem that links verified and executable code to cryptographic proofs. Jasmin code can be compiled to assembly for execution in production, and to EasyCrypt for proofs of functional correctness, cryptographic properties, and constant timeness. The toolchain has been used to prove indifferentiability, correctness, and constant timeness of a Jasmin SHA-3 implementation [Alm+19]. A tool taking the opposite approach of our cv2fstar is fs2cv [Bha+08a], extracting a CryptoVerif model from an F# protocol implementation. F# is a general-purpose programming language from the ML family and is part of the .NET Framework. An approach extracting CryptoVerif models from C via symbolic execution [AGJ12], started by extracting ProVerif models [AGJ11] and using a computational soundness result [BHU09].

PROOFS ON CODE. The Verified Software Toolchain (VST) Coq library has been used with FCF to verify the correctness and security of OpenSSL's HMAC implementation in C, using symbolic execution of the actual source code [Ber+15]. The F7 typechecker and its underlying methodology have been extended to support modular verification in the computational model of F# implementations [FKS11]. This allows to ensure asymptotic cryptographic guarantees by typing, and has been applied to the miTLS verified TLS implementation [Bha+13; Bha+14b]. The ideas have been further developed for use in F*, applied to the Record Layer of TLS 1.3 [DL+17]. [Bar+14] and [Gri+19] present frameworks for probabilistic relational verification in F*. Küsters, Truderung, and Graf present a framework to prove computational indistinguishability properties for Java-like programs [KTG12].

SYMBOLIC MODEL, EXTRACTING IMPLEMENTATIONS. Pironti and Sisto generate Java implementations from models written in spi calculus [PS10]. Sisto, Avalle, Pironti, and Pozza go further and implement spi calculus in Java, using Java both for modeling and implementation, and generating an implementation from the model with the help of annotations [Sis+11]. Chevalier and Rusinowitch generate OCaml implementations [CR10] from analyses done with CPPL [Gut+05]. Corin, Denielou, Fournet, Bhargavan, and Leifer show how to compile session types to cryptographic protocols in F# [Cor+08]. Fournet, Le Guernic, and Rezk present a methodology to generate F# implementations from information flow specifications [FGR09]. Noise Explorer generates ProVerif models as well as Go and Rust implementations from Noise protocol patterns [KNB19]. The symbolic analysis framework Dolev-Yao* (DY*) in F* "bridg[es] the gap between trace-based and type-based protocol analyses" [Bha+21], with executable models. Noise* [Ho+22] uses DY* in a development of a once-and-for-all verified generator of Noise protocol implementations, linking all the way down to efficient implementations in C.

[PM15] Petcher and Morrisett, "The Foundational Cryptography Framework"

[Alm+17] Almeida et al., "Jasmin: High-Assurance and High-Speed Cryptography"

[Bar+11] Barthe et al., "Computer-Aided Security Proofs for the Working Cryptographer"

[Alm+19] Almeida et al., "Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3"

[Bha+08a] Bhargavan et al., "Cryptographically verified implementations for TLS"

[AGJ12] Aizatulin et al., *Computational Verification of C Protocol Implementations by Symbolic Execution*

[AGJ11] Aizatulin et al., "Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution"

[BHU09] Backes et al., "CoSP: A General Framework for Computational Soundness Proofs"

[Ber+15] Beringer et al., "Verified Correctness and Security of OpenSSL HMAC"

[FKS11] Fournet et al., "Modular code-based cryptographic verification"

[Bha+13] Bhargavan et al., "Implementing TLS with Verified Cryptographic Security"

[Bha+14b] Bhargavan et al., "Proving the TLS Handshake Secure (As It Is)"

[DL+17] Delignat-Lavaud et al., "Implementing and Proving the TLS 1.3 Record Layer"

[Bar+14] Barthe et al., "Probabilistic Relational Verification for Cryptographic Implementations"

[Gri+19] Grimm et al., *A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations*

[KTG12] Küsters et al., "A Framework for the Cryptographic Verification of Java-Like Programs"

[PS10] Pironti and Sisto, "Provably correct Java implementations of Spi Calculus security protocols specifications"

[Sis+11] Sisto et al., "JavaSPI: A Framework for Security Protocol Implementation"

[CR10] Chevalier and Rusinowitch, "Compiling and securing cryptographic protocols"

[Gut+05] Guttman et al., "Programming Cryptographic Protocols"

[Cor+08] Corin et al., "A Secure Compiler for Session Abstractions"

[FGR09] Fournet et al., "A Security-Preserving Compiler for Distributed Programs"

EXTRACTING MODELS. Bhargavan, Fournet, Gordon, and Tse compile protocol implementations written in F# to ProVerif models [Bha+08b].

PROOFS ON CODE. The F7 tool has originally been developed to typecheck interfaces with refinement and dependent types for F# implementations, in a work by Bengtson, Bhargavan, Fournet, Gordon, and Maffeis [Ben+08]. The F5 typechecker is an advancement on the F7 work adding union, intersection, and polymorphic types [BHM14]. ASPIER [CD09] has been used to analyze authentication and secrecy properties of parts of the OpenSSL source code. Dupressoir, Gordon, Jürjens, and Naumann use the general-purpose verifier VCC and Coq to analyse C programs in the symbolic model [Dup+11].

[Bha+08b] Bhargavan et al., "Verified Interoperable Implementations of Security Protocols"

[Ben+08] Bengtson et al., *Refinement Types for Secure Implementations*

[BHM14] Backes et al., "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations"

[CD09] Chaki and Datta, "ASPIER: An Automated Framework for Verifying Security Protocol Implementations"

[Dup+11] Dupressoir et al., *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*

## 6.3 THE OUTPUT LANGUAGE

In this section, we briefly present some language elements of F*. The syntax of F* is mostly the same as in OCaml. F* supports algebraic data types, too. This includes product types like tuples and records, and sum types like tagged unions. Like OCaml, F* has pattern matching in let bindings and match statements, and supports higher-order functions.

For example, a custom type of natural numbers can be defined as follows in F*, as a tagged union:

```
type natural =
| Zero
| Succ: natural → natural
```

The constructor `Zero` does not have parameters, and `Succ` has one parameter, indicating the previous natural number. We can implement a predicate determining if an element of this type is non-zero with a match statement:

```
let nonzero n =
 match n with
 | Zero → false
 | Succ _ → true
```

Here, the underscore stands for an arbitrary value.

Comparing F* to OCaml, the main difference is that F*'s types are richer, and in particular can be dependent. The definition of a *dependent type* can depend on parameters. For example, using F*'s built-in type for natural numbers, `nat`, we can define a type of bounded natural numbers as follows:

```
type bounded l = n:nat{n < l}
```

The dependent type `bounded` has a parameter `l` that is used in the *refinement* `{n < l}` to define that this type containes all $n \in \mathbb{N}$ for which the condition $n < l$ holds. Generally, a refinement restricts the elements of the type to those that fulfill the predicate given inside the braces. Then, we can define a constant `t` as follows:

```
let t:bounded 3 = 2
```

Functions can also use dependent types, as in the following example for a subtraction function defined for our example type:

```
let bounded_sub
   (#l:nat)
   (a:bounded l)
   (b:bounded l{b ≤ a})
  : (bounded (l-b))
  = a - b
```

The function has three parameters: the bound `l`, a natural number `a` bounded by `l`, and a natural number `b` bounded by `l` that we additionally restrict with a refinement to being at most as large as `a`. In the function's return type, we use both the parameters `l` and `b`, to indicate that the result has a lower bound. The parameter `l` is prefixed with a `#`. This indicates that the parameter is implicit, which means that it can be omitted in case the F* typechecker can infer it from the remaining parameters. In this case, the value of `l` can be inferred from the types of both `a` and `b`.

Coming back to our example of a custom type for natural numbers, we can define the following decrease function to showcase pattern matching in let bindings:

```
let decr (n:natural{nonzero n}) =
  let Succ m = n in
  m
```

Here, we restrict the parameter to non-zero natural numbers using the predicate defined earlier. Then, F* understands that `n` can only be an element built with the `Succ` constructor of the tagged union, and we can use a let pattern matching to destruct it.

An important dependent type from the HACL* library that we use in cv2fstar is `lbytes len` from the HACL* module `Lib.ByteSequence`. This type implements fixed-length bytestrings, where `len` is the length in bytes. The parameter `len` is of type `size_nat`, which is the type of natural numbers up to a maximum of `max_size_nat` $= 2^{32} - 1$. The definition of the type `lbytes len` is as follows:

```
let lbytes (len:size_nat) =
  s:Seq.seq uint8{Seq.length s == len}
```

The type `Seq.seq` is from F*'s standard library and implements arbitrary-length sequences of a type given as parameter. Here, the parameter is `uint8`, which implements unsigned machine integers of size 8 bits.

F* functions can have pre- and post-conditions that the typechecker enforces. The subtraction function on bounded natural numbers can be written as follows, using pre- and post-conditions:

```
val bounded_sub2: l:nat → a:nat → b:nat
  → Pure (nat)
    (requires a < l ∧ b ≤ a)
    (ensures λc → c < l-b)

let bounded_sub2 l a b = a - b
```

Here, we use a function declaration with the `val` keyword. The `requires` clause states the pre-condition, and the `ensures` clause the post-condition that is a function of the return value of the function. With the effect `Pure`, F* checks that the function has no side effects and that it terminates. We did not indicate an effect for the previous functions. The standard effect in F* is

`Tot`, which is like `Pure`, just without pre- and post-conditions: it indicates a function without side effects that terminates.

Lemmas are a special kind of functions in F*, and can be used to record provable facts. Like with generic functions, different styles are available to write down lemmas and their pre- and post-conditions. We introduce the style that we use most in cv2fstar. Consider the following third implementation of our subtraction function:

```
let bounded_sub3 a b = a - b
```

Here, we removed all pre- and post-conditions, and refinements on types. We instead move them to a separate lemma:

```
let lemma_bounded_sub3
   (l:nat)
   (a:nat{a < l})
   (b:nat{b ≤ a})
  : Lemma (bounded_sub3 a b < l - b) = ()
```

The parameters of the lemma correspond to the parameters of our first implementation, and express the pre-conditions. The post-condition in parentheses behind the keyword `Lemma` expresses what we put into the result type or the post-condition of the other implementations of the function. F* is able to prove this lemma without further help, which is why the lemma's function body contains only `()`. Besides the interest in their own proof, lemmas can be used to guide F*'s typechecker in other proofs, by instantiating them with particular parameters, as needed.

To structure an F* development, it can be split into separate files, called *modules*. Other modules can be used within a module, by importing them, called *opening*. A module's interface and implementation can be separated into an `fsti` and `fst` file, respectively. By default, when a module `A` opens a module `B`, only the interface of `B` is visible inside `A`, while the implementation of `B` stays hidden. In F*, as a reminder, declarations without implementation begin with the `val` keyword. The implementation of a type begins with the `type` keyword, and implementations of functions, and proofs of lemmas begin with `let`. If an interface contains a `val` declaration but no implementation, the implementation must be provided in the `fst` file of the same module.

## 6.4 The Input Language

Starting from CryptoVerif's standard input language, we make two modifications: (1) We restrict it to a subset that is supported by cv2fstar; (2) We extend it to allow for annotations that guide the extraction.

### 6.4.1 *Annotations For Extraction*

There are two kinds of annotations. The first kind is marking blocks of code for extraction. Parts that shall be extracted must be wrapped in a block starting with `module_name {` and ending with `}`, and we call such a block a *module*. Each module will be extracted to a module in F*, with an `fsti` file for the interface and an `fst` file for the implementation.

The following code snippet shows an excerpt of the NSL case study as an example. Here, the `Initiator` module is defined inside the initiator process. We see that the oracle inside the process is replicated, through usage of the `foreach` keyword. We will discuss later that this replication is a requirement.

```
let initiator() =

 Initiator {

  foreach i_init ≤ Qinit do

  initiator_send_msg1 (...) :=
    ...
    return (...);

  ...

 }.
```

The second kind of annotations is for each type, constant, function, and table that is used within an extracted module. The required annotation differs for each of these:

type Types must be annotated with at least the name to be used in F*. If terms of the type are used in equality tests, then an indication of an equality test function is required.

There are many conditions under which an indication of a name for a serialization and deserialization function is required for a type. From CryptoVerif's point of view, it is required if terms of this type are cast to a bitstring, e. g., because they are assembled to a tuple or used directly within an out process (return of an oracle). From the point of view of the cv2fstar framework, a serialization function is also required if terms of this type appear in a table, an event, or a session entry: The framework defines printing helper functions that use the serialization function of a type to print tables, events, and session entries. We describe in Section 6.13.2 how we extract correctness assumptions on (de)serialization to F* lemmas.

If the type supports random sampling and the model does sample random values of this type, a function sampling random values of this type must be indicated. A type supports random sampling if it is not unbounded.

A specific kind of types receives special treatment: fixed types. These are types marked with `[fixed]` and model bitstrings of fixed length. For these types, all functions are predefined. The only annotation they require is their size in bits. The treatment of fixed types is covered in more detail in Section 6.6.

The predecessor project cv2ocaml allows the indication of a predicate function for a type. We explain in Section 6.14 why we do not need this in cv2fstar.

For a type $t$, the syntax of the annotations is as follows:

```
implementation type t="𝔾t(t)" [serial = "𝔾ser(t)","𝔾deser(t)";
                               equal = "𝔾eq(t)";
                               random = "𝔾$(t)"].
```

The function $\mathbb{G}_{\mathsf{t}}(t)$ indicates the F* name of a type $t$, $\mathbb{G}_{\mathsf{ser}}(t)$ and $\mathbb{G}_{\mathsf{deser}}(t)$ the names of the serialization and deserialization functions of a type $t$, $\mathbb{G}_{\mathsf{eq}}(t)$ the name of the equality test function of a type $t$, and $\mathbb{G}_{\$}(t)$ the name of the random sampling function of a type $t$.

For a fixed type $t$, the syntax for its size annotation is as follows, where the function $\mathbb{G}_{\mathsf{size}}(t)$ indicates the size of a fixed type $t$ in bits:

```
implementation type t="𝔾_size(t)".
```

The notation style with functions $\mathbb{G}_{\cdot}(\cdot)$ is inspired by the style used in the cv2ocaml publications, in particular [CB13].

[CB13] Cadé and Blanchet, "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH"

constant Constants must be annotated with their name in F*. For a constant $c$, the annotation's syntax is as follows, where $\mathbb{G}_{\mathsf{c}}(c)$ indicates the F* name of a constant $c$:

```
implementation const c="𝔾_c(c)".
```

function Functions must be annotated with at least their name in F*. Functions declared with `[data]` in a CryptoVerif model have an efficiently computable inverse function. For such a function, and if the inverse is used in the model, an indication of the name of a function implementing the inverse is required. The inverse of a function is used if it appears on the left side of the equal sign of a pattern matching. The syntax for these annotations is, for a function $f$:

```
implementation fun f="𝔾_f(f)" [inverse = "𝔾_inv(f)"].
```

The function $\mathbb{G}_{\mathsf{f}}(f)$ indicates the F* name of a function $f$, $\mathbb{G}_{\mathsf{inv}}(f)$ indicates the name of the inverse of $f$. We describe in Section 6.13.2 how we extract correctness assumptions on inverses to F* lemmas.

It is allowed to extract letfuns to a function call in F*, thereby hiding the body of the letfun as it is defined in the CryptoVerif model. Letfuns in CryptoVerif can contain random sampling of variables, and thus, such functions require access to entropy. For such a letfun, it is required to indicate this by a `use_entropy` annotation:

```
implementation fun f="𝔾_f(f)" [use_entropy = "true"].
```

The handling of entropy is covered in more detail in Sections 6.6.1 and 6.8.

table Tables must be annotated with the name that shall be used in the algebraic data types that implement them in F*. For a table tbl, the annotation is as follows:

```
implementation table tbl="𝔾_tbl(tbl)";
```

### 6.4.2 *Restrictions*

The CryptoVerif code of modules marked for extraction is not allowed to contain `find` terms. The reason for this is mostly to avoid the complexity that supporting `find` would introduce for a soundness proof. While we are not providing a soundness proof for cv2fstar in this work, a proof has been done for cv2ocaml [CB15], and that work had shown that restricting the input language by disallowing `find` makes the problem much more tractable. For

[CB15] Cadé and Blanchet, "Proved Generation of Implementations from Computationally Secure Protocol Specifications"

cv2fstar, we decided to inherit this restriction. Both cv2ocaml and cv2fstar support tables, which are a specific form of `find`, and we believe tables should be sufficient to model practically interesting protocols. One might also think that the semantics of `find` is dubious for practical implementations: While read access to variables of other sessions within the same module is straightforward, read access to variables of other modules that might be executed on remote machines is not. However, it would be possible to implement such access, assuming a secure communication channel between remote modules.

Modules marked for extraction are not allowed to contain `event_abort`. The semantics of `event_abort` in CryptoVerif is that the entire game stops as soon as this event is dispatched. In a practical implementation, it is not feasible to abort the entire system: The module that reached an `event_abort` would need to communicate this fact to other modules, and the adversary could easily suppress this.

We restrict the model to a specific top-level structure. It must be a parallel composition of processes under replication, and each process must be extracted as a module, as follows:

```
(
 module_1 {
  foreach i_1 ≤ N_1 do
   Oracle_1 (...) := ...
    return (...)
   ...
 }
|
 module_2 {
  foreach i_2 ≤ N_2 do
   Oracle_2 (...) := ...
    return (...)
   ...
 }
|
 ...
)
```

This restriction is motivated as follows: Extracted to a function in an F* module, the top-level oracle of a module can be called at any time, and so we do not allow any model code above it (because we would not be able to enforce that this other code is executed before); also, it can be called any number of times, and so we require that it is under replication in the model (effectively preventing an application from calling the top-level function more than once is hard). Thus, it seems sensible to require the CryptoVerif models and thus the cryptographic proof to be written in a way that accounts for that. The restriction to this specific top-level structure also means that all parts of the model must be extracted. The motivation behind this is that an adversary against the F* code should be able to do everything that an adversary is allowed to do in the CryptoVerif model.

This restriction of the top-level structure means that, in particular, cv2fstar does not allow a non-replicated setup oracle, like it is common in CryptoVerif and ProVerif models. Such an oracle is for example often used to sample a key for a hash function or a random oracle; this key just models the choice of the hash function and could be ignored in an implementation that uses a

specific hash function. In the top-level structure imposed by cv2fstar, this could be implemented by a replicated setup oracle that stores the sampled key in a table; oracles in other modules then need to retrieve it from the table and yield if the adversary did not yet call the setup oracle.

Our usage of HACL* on the F* side introduces some restrictions, too: First, types declared as fixed-length in CryptoVerif are restricted to at most $2^{32}-1$ bytes. Second, technically, we translate CryptoVerif bitstrings, including fixed-length types, to HACL* *byte*strings. This affects practical systems that need bitstrings with a length that is not a multiple of eight. Such applications would need to work around this, or extend the framework accordingly, for example by using the `uint1` type instead of the `uint8` type underlying `bytes`. This woud entail rewriting equality test functions for bitstrings instead of bytestrings in HACL*.

Some restrictions that are present in cv2ocaml are no longer present in cv2fstar. Events are extracted to F*, while they are ignored in cv2ocaml. We allow replication indices to occur in events under the condition that the entire hierarchy of replication indices is used together in a tuple. This is because we use a single session identifier in the cv2fstar framework, and do not translate the replication indices individually. The translation of replication indices in events is covered in more detail in Section 6.11.

## 6.5 Translating to F*: An Overview

In the following sections, we describe step-by-step how we translate CryptoVerif models to executable F* specifications. We say *executable* because the specifications we write are extractable to OCaml code via F*'s built-in toolchain, and that is also how we execute the code of our case study presented in Section 6.15.

The code generated by the cv2fstar compiler does not stand on its own, but runs in the context of what we call the *cv2fstar framework*. This framework consists of two parts: First, a collection of F* modules that define the generic behaviour of tables (Section 6.9), sessions (Section 6.12), events (Section 6.10), and entropy (Section 6.6.1). This concerns the F* modules `State`, `NatMap`, `Helper` and `RandomHelper`. These form what we call the *state*, see Section 6.8. Second, an F* module `CVTypes` containing definitions of standard CryptoVerif types and functions, see Section 6.6.

The cv2fstar compiler generates several model-specific files from a CryptoVerif model. In the following, we use $\mathbb{M}$ to refer to model's name in uppercase letters, derived from the filename of the CryptoVerif model. It is used in the names of the generated F* modules. We use $\underline{m}$ to refer to the all-lowercase version, used in type names. The compiler generates F* interface files with file extension `fsti` and for some modules, F* implementation files with file extension `fst`. The compiler generates the following model-specific files. They all depend on `CVTypes`, and we indicate additional dependencies in the following. Figure 6.1 gives a visual overview of how the different parts of a CryptoVerif model translation come together with cv2fstar.

$\mathbb{M}$`.Types.fsti`: This file is generated with the declarations of types that are part of the CryptoVerif model, as discussed in Section 6.6.2. It depends

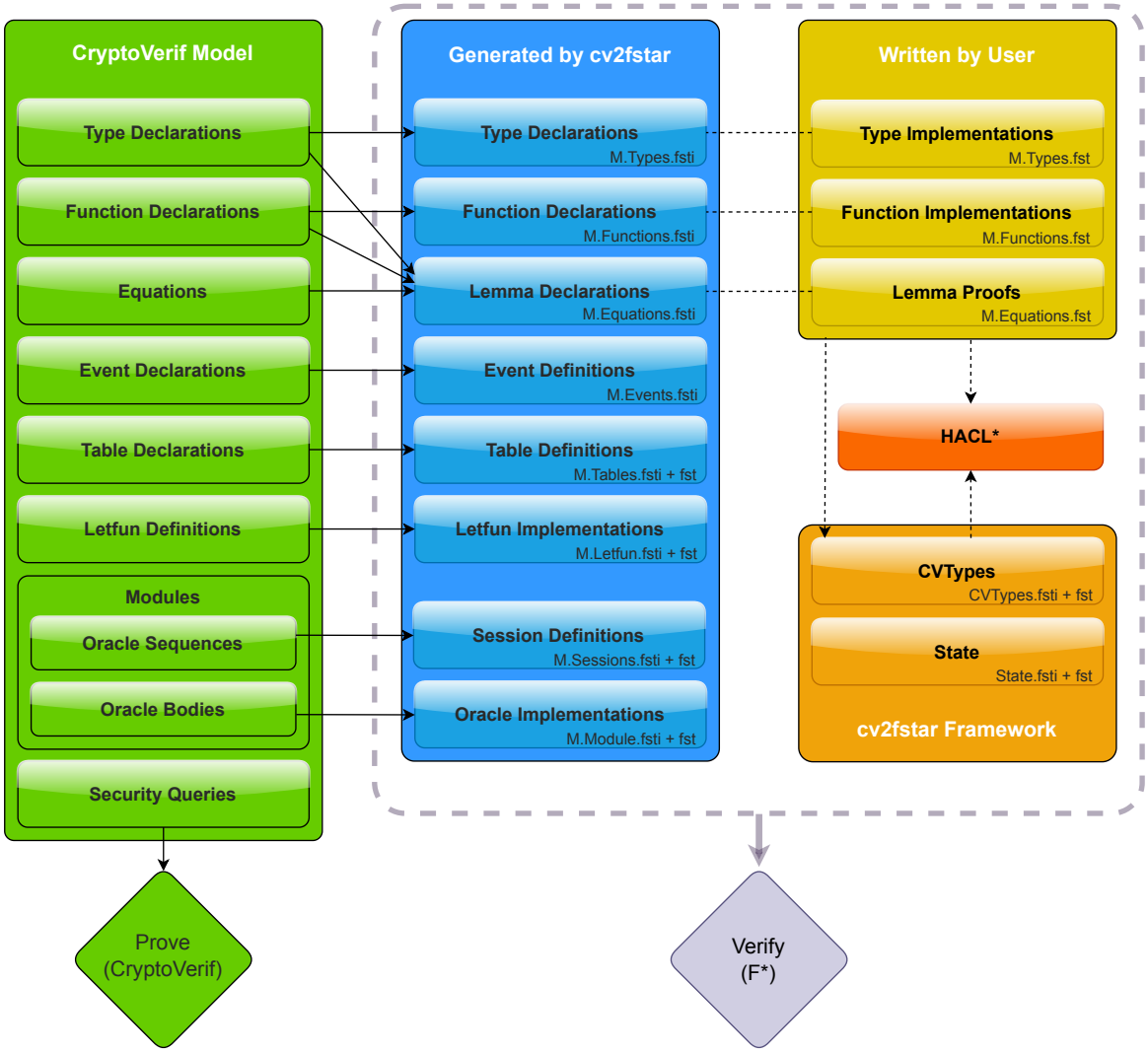



FIGURE 6.1: Visual overview of how the different parts of a cv2fstar translation project come together. The uppercase letter M stands for $\mathbb{M}$, the uppercase name of the CryptoVerif model.
On the left in green, we show a CryptoVerif model and its parts. The solid box in the backround of the entire column represents the fact that all these parts are within one CryptoVerif model file. Module block annotations contain oracle sequences, and each oracle has a body, which is why we regroup them as modules. The security queries stated in the model are proved with CryptoVerif.
On the middle in blue, we show the files generated by the cv2fstar compiler. An arrow coming from the left means that this part from the CryptoVerif model contains information needed to generate the part in the middle. For oracle implementations, separate `fsti` and `fst` files are generated for each module, using the name of the module block annotation as F* module name. The only part from the left that is not used for any generation is the security queries.
On the right in yellow, we show the parts that a user has to manually implement to complete an extracted F* specification: types, functions, and proofs of lemmas. This implementation relationship is represented by dotted lines between middle and right. We do not show a user-written application that uses the extracted executable protocol specification.
On the right at the bottom in orange, we show the cv2fstar framework that is written and verified once and for all. The two most important parts are `CVTypes` implementing CryptoVerif's built-in types in F*, and `State`, providing functions for managing events, tables, sessions, and entropy.
On the right in the middle in red, we show the HACL* library that is used by `CVTypes`, and that can be used by the user-written implementations. This relationship is represented by dotted arrows. Likewise, the user-written implementations can use `CVTypes`. The event, table, and session definitions use `State` and other modules visible in the diagram; we do not display these dependencies because this would introduce too many arrows.
The entire F* development shown in the middle and the right column is verified using the F* typechecker.

on CVTypes. The user has to provide implementations for the types in a manually written file M.Types.fst.

M.Functions.fsti: This file is generated with the declarations of functions and constants, as presented in Section 6.7. It depends on M.Types and CVTypes. The user has to provide implementations for them in a manually written file M.Functions.fst.

M.Equations.fsti: This file is generated with lemmas expressing the non-cryptographic assumptions made in the CryptoVerif model as described in Section 6.13. It depends on M.Types, M.Functions, and CVTypes. The user has to provide proofs for them in a manually written file M.Equations.fst.

M.Events.fsti AND fst: These are the generated files for the types and functions needed for events, as presented in Section 6.10 and by example in Section 6.15.6. It depends on M.Types, State, and CVTypes.

M.Tables.fsti AND fst: These are the generated files for the types and functions needed for tables, as presented in Section 6.9 and by example in Section 6.15.3. It depends on M.Types, State, and CVTypes.

M.Sessions.fsti AND fst: These are the generated files for the types and functions needed for sessions, as described in Section 6.12.4 and by example in Section 6.15.7. They depend on M.Types, State, and CVTypes.

M.Protocol.fst: This small generated file defines the model-specific state type *m*_state. It depends on M.Types, M.Tables, M.Sessions, M.Events, and State.

M.Letfun.fsti AND fst: These are the generated files for the interfaces of letfun functions and their implementations, as described at the end of Section 6.11. They depend on M.Types, M.Functions, M.Tables, M.Sessions, M.Events, M.Protocol, State, Helper, and CVTypes.

M.Module.fsti AND fst: For each module block annotation, an interface and an implementation file is generated for the sequence of oracles contained in the module, as described in Section 6.12 and by example in Section 6.15.7. The name used in the module block annotation is used instead of Module, in the filename. They depend on M.Types, M.Functions, M.Tables, M.Sessions, M.Events, M.Protocol, State, Helper, and CVTypes.

## 6.6 TRANSLATING TYPES

A type *t* is declared as follows in a CryptoVerif model:

```
type t.
```

A type can be declared with options *o*:

```
type t [o].
```

The most common options are fixed or bounded to indicate fixed-length bitstrings, or bitstrings with a maximum length; and large to indicate that the number of elements of this type is large enough such that the probability

that two randomly sampled elements collide is negligible. If two options are used, they are separated by a comma. For more options, we refer to the CryptoVerif manual that is available in the CryptoVerif download. [6]

[6] https://cryptoverif.inria.fr

### 6.6.1 *Built-In and Fixed-Length Types*

CryptoVerif has three built-in types that are prominently visible: `bitstring`, `bitstringbot`, and `bool`. A type that is not immediately visible as such is the one of replication indices. CryptoVerif uses positive integers which we translate to the built-in `nat` in F*. CryptoVerif's `bool` is translated to F*'s built-in type `bool`. CryptoVerif's `bitstring` is unbounded and we map it to HACL*'s `bytes` which is unbounded as well. It is defined in the module `Lib.ByteSequence`. An important kind of not-built-in types in CryptoVerif is the type of fixed-length bitstrings, annotated with the option `[fixed]` in type declarations. We mention it here already because it is translated in a pre-defined way, with limited choice by the author of the CryptoVerif model. Fixed types are translated to `lbytes len` from HACL*'s module `Lib.ByteSequence`, where `len` is the length of the bitstring in bytes, with a maximum of `max_size_nat` $= 2^{32} - 1$ bytes. This means fixed-length types are limited to this many bytes. This should be enough for any current practical purpose but is, in theory, a restriction compared to CryptoVerif's semantics where there is no such limit. The length is the only parameter that the CryptoVerif model author can choose for a fixed type. As an example, a `fixed` CryptoVerif type annotated with a size of 64 bits is translated to `lbytes 8`; this should explain why no name annotation is required for `fixed` types.

CryptoVerif's type `bitstringbot` differs from `bitstring` only in that it adds a failure symbol `bottom`. We translate it as option type `option bytes`, where `bottom` is constructed by `None` on the F* side, and a bitstring `b` by `Some b`. [7]

[7] An `option` type is a particular *datatype*, *sum type*, or *tagged union* predefined in F*. The type `option a` with type `a` as parameter has two constructors: `None`, and `Some a`.

EQUALITY TEST FUNCTIONS FOR BUILT-IN TYPES. HACL*'s `Lib.ByteSequence` implements a constant-time equality test function `lbytes_eq` for `lbytes len`. We use it for CryptoVerif's fixed-length types, wrapped into a function named `eq_lbytes` for reasons of consistency with the remainder of the framework.

HACL* does not provide an equality test function for the unbounded type `bytes`. However, CryptoVerif *does* support equality tests on `bitstring`, and so we wrote our own equality test function `eq_bytes` for the type `bytes` that uses `lbytes_eq` internally for chunks smaller or equal to `max_size_nat` in size. For bytestrings shorter or equal to $2^{32} - 1$ bytes, it directly falls back to `lbytes_eq`.

The equality test function `eq_obytes` for `bitstringbot` is straightforward and returns true either if both `bitstringbot` are `bottom` or if the `bitstrings` are equal. For `nat` and `bool`, we use F*'s built-in operator for decidable equality `(=)` without wrapper.

CryptoVerif makes several assumptions about equality test functions. One of them is commutativity, and we discuss in Section 6.13 how we translate this as F* lemma and how we prove it.

For the pre-defined types we go further: We prove that `eq_bytes` and `eq_obytes` are equivalent to F*'s built-in operator for propositional equality `(==)` for all pairs of parameters (bytestrings and optional bytestrings). The operator `(==)` does not extract to executable code because propositional equality is undecidable in F*, it is only usable in logical statements. [8] The interesting property for us is that it is an equivalence relation. Thus, by our proofs, we establish that `eq_bytes` and `eq_obytes` are equivalence relations, respectively.

[8] For more details on this, see https://www.fstar-lang.org/tutorial/book/part1/part1_polymorphism.html#equality

SERIALIZATION AND DESERIALIZATION OF BUILT-IN TYPES. Generally, a serialization function in cv2fstar returns `bytes`, and a deserialization function takes `bytes` as input and returns an option type of the original type. It returns `None` if the input cannot be parsed as an instance of the original type.

Like any type, built-in types can be used within CryptoVerif tuples or as returned term of an oracle, and so we need to define serialization and deserialization functions for built-in types. The target type of serialization is the image of `bitstring`, which is `bytes`. For `bytes` itself, serialization is the identity function and so we do not spell it out explicitly in `CVTypes`. For `lbytes len`, serialization is a straightforward type cast to `bytes`. However, for deserialization, we require the expected length `len` of the resulting `lbytes len` as a parameter. We return `None` in case the length given as parameter and the actual length of the input bytes do not match. Apart from that, it is a straightforward type cast.

For the serialization of `bitstringbot` (`option bytes` in F*), we express `bottom` (`None`) as a null byte `0x00`, and any bitstring as the byte `0x01` concatenated with the actual bytestring.

For `nat`, we use HACL*'s serialization and deserialization functions `nat_to_bytes_be` and `nat_from_bytes_be` from `Lib.ByteSequence` which encode a `nat` as a big-endian byte sequence. The choice of big-endian is arbitrary but matches the usually used network encoding.

For `bool`, we chose a simple encoding of `false` as `0x00` and `true` as `0xFF`. One bit would be enough, however, the smallest unit of HACL*'s `bytes` is one byte, as `bytes` is implemented as a sequence of bytes. For the sake of not increasing the complexity of the framework, we did not strive for a more efficient serialization of `bool`. As an optimization for future work, the target for serialization could be made a type of bitstrings, based on the `uint1` type instead of the `uint8` type underlying `bytes`.

We prove two correctness lemmas for each pair of (de)serialization functions. It is the same kind of lemmas that cv2fstar generates as proof obligation for user-defined types that have serialization and deserialization functions. This is described in more detail in Section 6.13.

PRINTING HELPER FUNCTIONS OF BUILT-IN TYPES. As helper functions for debugging purposes, we define printing functions for the built-in types, and most importantly for `bytes`. The print function for `bytes` is used for printing of user-defined types via the type's serialization function. This way, the user is only required to implement serialization (and deserialization) and gets printing helpers for free. The print function for `bytes` prints a hexadecimal

Listing 6.1: Interface for entropy in the cv2fstar framework, as defined in `CVTypes.fsti`.

```
1 val entropy:Type0
2
3 val initialize_entropy: Lib.RandomSequence.entropy → entropy
4 val finalize_entropy: entropy → Lib.RandomSequence.entropy
5
6 val gen_nat: max: nat{max > 0} → entropy
7   → entropy * n: nat{n ≤ max}
8 val gen_lbytes: l: Lib.IntTypes.size_nat → entropy
9   → entropy * lbytes l
```

representation of a given bytestring.

ENTROPY, AND RANDOM SAMPLING FOR BUILT-IN TYPES. In cv2fstar, all functions are pure by design, i. e., they terminate and are in particular without side effects. If a functions needs to randomly sample a value, we need to explicitly pass the randomness or at least some representation of entropy to it. In CryptoVerif, random sampling can be done in terms with the `<-R` syntax, and it is done implicitly when multiple entries match a table get query, to (almost) uniformly random choose one of the matches.

For cv2fstar, we decided to go for an entropy-passing style as follows: At the initialization of the framework, some initial entropy must be passed to the framework. The framework keeps the entropy value and passes it to each function that does random sampling. Such a function internally takes some entropy from the entropy pool, and returns the new state of the entropy pool along with the actual result of the function. The cv2fstar framework provides functions for random sampling of built-in types that work this way – they take entropy as parameter, and return entropy and a randomly sampled value. This style avoids side effects and gives us a pure specification. We will discuss in a moment that the actual implementation is not necessarily free from side effects.

There are two layers that we describe in the following. First, we discuss the interface defined for entropy in cv2fstar's module `CVTypes.fsti`. It defines entropy as an abstract type. Second, we explain the implementation of entropy in `CVTypes.fst`, which instantiates entropy and random sampling with specific algorithms. As with all well-designed abstractions, this allows us to replace the implementation of random sampling later, without touching the rest of the framework. We will discuss related future work at the end of this section.

Listing 6.1 shows how the interface of entropy and random sampling is defined in `CVTypes.fsti`. Line 1 shows the definition of entropy as an abstract type. For this first version of cv2fstar, we decided to require the entropy variable to be initialised once at the beginning when the framework is set up. The function used for this is shown in Line 3. It takes a value of a specific type of entropy as parameter, and returns the framework's abstract entropy type – which might or might not be the same type, depending on the implementation. This specific type of entropy is HACL*'s entropy as defined in the module `Lib.RandomSequence`. At the end of execution of the framework, the `finalize_entropy` function (Line 4) can be called to

retrieve the new value of the HACL⋆ type of entropy, to close the circuit.

CryptoVerif samples almost uniformly random integers to choose a table entry among multiple query matches. We define `gen_nat` to support this on the F⋆ side (Line 7). The function takes a `nat` parameter to define the interval $[0, m]$ in which to sample the integer.

There is no random sampling for `bitstring` in CryptoVerif because this type is unbounded. CryptoVerif allows random sampling for types that are annotated with `[bounded]` or `[fixed]`. cv2fstar currently supports random sampling for fixed types with `gen_lbytes` (Line 9). This function takes the desired length in bytes as parameter.

The implementation of the type `entropy` and of the functions `initialize_entropy`, `finalize_entropy`, `gen_nat`, and `gen_lbytes` is outsourced to the cv2fstar module `RandomHelper`. Its interface defines the same type and functions. We discuss the implementation in the following.

```
1 noeq type entropy =
2   | Entropy : system_entropy: Lib.RandomSequence.entropy → entropy
3
4 let initialize_entropy ent =
5   Entropy ent
6
7 let finalize_entropy ent =
8   let Entropy entropy = ent in
9   entropy
```

In Line 2, we define entropy as a type with a single constructor `Entropy` that takes a value of HACL⋆'s entropy type as parameter. [9] Initialization simply creates an instance of entropy with the entropy value passed to it, and finalization unwraps and returns it. Here, it becomes clear that in our current implementation, we just use the same entropy type internally.

For the actual random sampling, we use the `crypto_random` function from HACL⋆'s `Lib.RandomSequence` module. It is the only module for random sampling that HACL⋆ offers for the pure world of specifications written in F⋆. [10] Only the *interface* of `Lib.RandomSequence` is written in F⋆:

```
1 val entropy: Type0
2 val entropy0: entropy
3 val crypto_random: entropy → len:size_nat-> entropy * lbytes len
```

The implementation in a directly executable language has to be linked after extraction of the F⋆ code. As we extract our case study's code to OCaml for execution, we briefly look at the OCaml implementation: It instantiates `entropy` as an OCaml integer and uses `0` as initial value. The function `crypto_random` internally uses the OCaml crypto library Cryptokit's function `Random.hardware_rng` that uses the RDRAND CPU instruction to get random bytes[11]. As new entropy value, it always returns `0`. At this point it becomes clear that the entropy-passing style is of merely philosophical interest when using such an implementation as backend. However, the interface defined by `Lib.RandomSequence` is also not giving any information about how the two entropy values relate, and how the returned randomness relates to them. Concluding this brief look to the low-level implementation of HACL⋆'s `crypto_random`, we now look at how we use it to sample random bitstrings of fixed length.

[9] The keyword `noeq` prevents F⋆ from generating the decidable equality operator `(=)` for this type, which would otherwise fail with an error.

[10] For efficient implementations in Low⋆, there is the library `Lib.RandomBuffer.System` for random sampling.

[11] This code can be found in the file `lib/ml/Lib_RandomSequence.ml` in HACL⋆.

The implementation of `gen_lbytes` is straightforward, see Listing 6.2: We unwrap the entropy value, and then call `crypto_random` with it and the desired length of the bitstring. We construct the new entropy value and return it along the random bitstring.

Listing 6.2: Implementation of random sampling for fixed-length bitstrings.

```
let gen_lbytes num ent =
  let Entropy entropy = ent in
  let entropy, bytes = crypto_random entropy num in
  Entropy entropy, bytes
```

Randomly sampling a natural number within an interval $[0, m]$ is more involved, given that we only have `crypto_random` which generates a bytestring of length at most `max_size_nat`: First, we need to generate an almost uniformly random natural number from a uniformly random bytestring. Second, we need to support this for an arbitrary large $m$. The second problem is of course mostly a problem of theoretical interest, because most practical systems will not have table queries that match this many entries. We implement the function `crypto_random_unbounded` to solve this – a variant of `crypto_random` that takes a natural number $n$ as parameter and that calls `crypto_random` internally as often as necessary to produce a bytestring of length $n$. In the implementation of `gen_nat`, we first compute the minimum number of bytes $n'$ necessary to represent the upper inclusive bound $m$ of the interval:

$$n' = \left\lceil \frac{1 + \log_2 m}{8} \right\rceil .$$

We then add an accuracy parameter $k$ to it to compute the number of random bytes $n$ that we are going to sample:

$$n = n' + k .$$

A higher value for $k$ reduces the skew in the almost uniform distribution. We use `crypto_random_unbounded` to generate $n$ random bytes, and transform them to a natural number $r$ using a built-in HACL⋆ function. Finally, we compute the almost uniformly random number in $[0, m]$ by computing $r$ mod $(m+1)$. The parameter $k$ is configurable in cv2fstar's code, we set it to 10 for our case study.

Concerning future work on random sampling, we plan to replace the use of `crypto_random`. Currently, we use the hardware random number generator for each random byte. This is less efficient than using a cryptographically secure pseudo-random number generator (CSPRNG) and seeding it in reasonable intervals from the hardware random number generator, or any other high-quality entropy pool offered by the operating system. As a teaser, with the methodology presented above, the implementation of the entropy could look like this, because we need to keep the state of the CSPRNG:

```
noeq type entropy =
  | Entropy :
     system_entropy: Lib.RandomSequence.entropy
    → Spec.HMAC_DRBG.state Spec.Agile.Hash.SHA2_512
    → entropy
```

The CSPRNG HMAC-DRBG is implemented in HACL⋆, and it is parametrized by a hash function. We would initialize this state within `initialize_entropy` and reseed it from the system's entropy pool whenever necessary. Here, the entropy type's constructor takes two parameters, and it becomes clearer why the use of an explicit constructor makes code more readable: in contrast to simply using a tuple of two elements (e. g. `(a, b)`), the explicit constructor makes it clear from reading alone that we are dealing with entropy (e. g. `Entropy a b`).

### 6.6.2 *Other Types*

Now that the treatment of built-in and fixed types has been described, we look at how other, user-defined types are translated. cv2fstar only translates type declarations for them, and if required and indicated by annotations, declarations of equality test, serialization and deserialization, and random sampling functions. For a type $t$, the following code is generated in $\mathbb{M}$`.Types.fsti`:

```
val 𝔾ₜ(t): Type0
val 𝔾_eq(t): 𝔾ₜ(t) → 𝔾ₜ(t) → bool
val 𝔾_ser(t): 𝔾ₜ(t) → bytes
val 𝔾_deser(t): bytes → option (𝔾ₜ(t))
val 𝔾_$(t): entropy → entropy * 𝔾ₜ(t)
```

The user needs to write a file $\mathbb{M}$`.Types.fst` with appropriate implementations of the type and its functions.

## 6.7 Translating Functions and Constants

Constants. A constant $c$ of type $t$ is declared as follows in a CryptoVerif model:

```
const c: t.
```

This declaration is translated in $\mathbb{M}$`.Functions.fsti` as follows:

```
val 𝔾_c(c): 𝔾ₜ(t)
```

Functions. A function $f$ with parameters of types $t_i, 1 \leq i \leq n$ and return type $t$ is declared as follows in a CryptoVerif model:

```
fun f(t₁, ..., tₙ): t.
```

The optional `[data]` keyword to indicate that $f$ has an efficiently computable inverse can be added at the end:

```
fun f(t₁, ..., tₙ): t [data].
```

The function declaration in F⋆ is generated in the following way:

```
val 𝔾_f(f): 𝔾ₜ(t₁) → ... → 𝔾ₜ(tₙ) → 𝔾ₜ(t)
```

If the function is declared `[data]`, and the inverse of the function is used in the model, an additional function declaration for the inverse is generated:

```
val 𝔾_inv(f): 𝔾ₜ(t) → option (𝔾ₜ(t₁) * ... * 𝔾ₜ(tₙ))
```

The return type is an option type because not each element of type $t$ might be invertible, i. e., not each element of type $t$ might be in the image of $f$.

## 6.8 The State Type

In Section 6.6, we introduced entropy as a kind of state of which the cv2fstar framework keeps track during its execution. In the following sections, we introduce other kinds of such state. Entropy is generic and not specific to a CryptoVerif model. There are three data types that are specific to a CryptoVerif model and important for the state of the execution: tables, events, and sessions. We introduce them step by step in Section 6.9 (tables), Section 6.10 (events), and Section 6.12 (sessions). What we want to say already now is that we group these three data types into a polymorphic `state_type`. It looks as follows, where we use the data types that we introduce later:

```
noeq type state_type =
  | StateType :
      tt: table_type
    → st: session_type
    → event_type: Type0
    → state_type
```

We define getters `tt_of`, `st_of`, and `et_of` to access the individual data types of tables, sessions, and events, given a `state_type`. The cv2fstar compiler generates code to instantiate `state_type` for a CryptoVerif model, as described in the following sections.

With this, we define a polymorphic type `state` that is parametrized by a `state_type`. An instance of this type holds the state for each of the three model-specific data types (tables, sessions, events), and for the entropy pool:

```
noeq type state (stt: state_type) =
| State :
    ent: entropy →
    tabs: tables (tt_of stt) →
    sess: sessions (st_of stt) →
    evs: list (et_of stt)
  → state stt
```

Internally, the framework provides getters and setters for the state's fields as helper functions.[12] The getters are called `entropy_of`, `tables_of`, `sessions_of`, and `events_of`, and take a `state` as parameter. The setters are called `state_upd_entropy`, `state_upd_tables`, `state_upd_sessions`, and `state_upd_events`. They take a state and a new entropy, tables, sessions, or events object as parameter, and return a new state with the appropriate field replaced by the new object, and the other fields unchanged.

[12] In this case, internally means, that they are not exposed in the interface `State.fsti` but are only present in `State.fst`.

The framework's idea concerning the state is that the *outside* interface does not allow direct manipulation of state fields. Changes are only possible via functions exposed in `State.fsti`. This way, on the one hand, we hide the implementation details of the state's components, and on the other hand, we control what changes can be made, and can ensure that they are sound.

In Section 6.6, we introduced our handling of entropy and random sampling. An important function exposed by the framework in this context is `call_with_entropy`. We use it as a wrapper to provide functions with entropy. The function `call_with_entropy` takes two parameters: the state, and a function to be wrapped. This function must take entropy as input and return a pair of entropy and some other return value. The output of

`call_with_entropy` is a new state and the return value of the wrapped function. The interface is as follows:

```
val call_with_entropy:
   #stt: state_type →
   #a: Type→
   st0: state stt →
   (entropy → entropy * a)
  → state stt * a
```

The function has two implicit parameters, marked with #, the state type and the return type of the wrapped function. In most cases, F* should be able to infer them from the next two parameters, the state and the wrapped function. Then, it is not necessary to provide the implicit parameters. This function is also a good example for a dependent type: Its return type `state stt * a` depends on the parameter `stt`. The implementation of the function is as follows:

```
let call_with_entropy st0 f =
  let ent0 = entropy_of st0 in
  let ent1, res = f ent0 in
  let st1 = state_upd_entropy st0 ent1 in
  st1, res
```

This wrapper is used for any random sampling with `gen_nat` and `gen_lbytes`.

## 6.9 Translating Tables

As a start, a quick recap about tables in CryptoVerif. In a CryptoVerif model, tables are declared using

```
table tbl(t_1, ..., t_n).
```

where `tbl` is the table name and $t_1$ to $t_n$ are the types of an entry's fields. In a process, entries can be added to a table by a term

```
insert tbl(M_1, ..., M_n); M
```

where $M_1$ to $M_n$ must be terms with types matching the types of the table declaration. A table insert always succeeds in CryptoVerif, so there is no branching, and only one next term $M$ that is evaluated. Entries cannot be removed from tables.

A table is queried by a term

```
get tbl(p_1, ..., p_n) suchthat M in M' else M''
```

where $p_1$ to $p_n$ are patterns. The query tries to find an entry with fields matching the patterns such that the term $M$ is `true`. If exactly one matching entry is found, the term evaluates to the result of $M'$. If more than one matching entries are found, one among them is chosen almost uniformly random, and the term evaluates to the result of $M'$. Otherwise, it evaluates to the result of $M''$. The keyword `[unique]` can be added directly after `get` to make it `get[unique]`. Then, CryptoVerif tries to prove that there is only ever at most one matching table entry for this query, up to negligible probability.

In the following, we describe by which types CryptoVerif tables are expressed in the cv2fstar framework. Then, we line out how `insert` and the different variants of `get` are implemented. As a motivational overview, let us discuss what we need for tables. We want to identify tables by their name,

TABLE 6.1: Common abbreviations used in the cv2fstar framework.

| Abbreviation | Meaning |
|---|---|
| tt | table type |
| tn | table name |
| tns | table names |
| te | table entry |
| tft | table filter type |
| rt | return type |
| st | session type |
| se | session entry |
| on | oracle name |
| et | event type |

as in CryptoVerif; for each table, we need a data structure containing the entries, and also a data type for the tables's entries.

We begin with a generic type for tables, `table_type`. It has one constructor, parametrized by two values: a type meant to enumerate table names, and a function type meant to map a table's name to the type of its entries.

```
noeq type table_type =
| TableType :
   tn: eqtype (* table names *) →
   te: (tn → Type0) (* table entries *) → table_type
```

We define two getters: `table_name` to retrieve the type enumerating table names, and `table_entry` to retrieve the type of a table's entries, given a `table_type`.

We introduce a polymorphic type `table`, parametrized by a table type and a table name. A `table` is implemented as a sequence of table entries of the appropriate type:

```
type table (#tt: table_type) (tn: table_name tt) =
  Seq.seq (table_entry #tt tn)
```

We group all tables of a model in a type `tables` parametrized by the table type. We use a function-type mapping from table name to a table: [13]

```
type tables (tt: table_type) = tn: table_name tt → table #tt tn
```

[13] This could be implemented more efficiently with a record, as the set of tables in a model is fixed. We scheduled this change for immediate future work.

For the initialisation of the framework, we provide a function `init_tables` that constructs an empty instance of `tables` for a given table type: it returns a function that maps each table name to the empty sequence. Eventually, we want to implement a functionality to write tables to disk and to read them from disk to allow for persistence. In this first version of cv2fstar, the framework always starts with empty tables. The implementation of the types `table_type`, `table_name`, and `table_entry` is model-specific; we will discuss the NSL example in Section 6.15.

We now describe the implementation of `insert` and `get`. Beginning with `insert`: it takes a state, a table name, and a table entry to be added to the table as parameters, and returns a new state with the tables field updated accordingly. The function declaration is as follows:

```
val insert:
   #stt: state_type →
   st0: state stt →
   tn: table_name (tt_of stt) →
   table_entry #(tt_of stt) tn
  → state stt
```

The matching condition of a CryptoVerif `get` query consists of the patterns (one for each field), and an additional condition after the `suchthat` keyword. In cv2fstar, we express these by *one* filter function that is provided as a parameter to `get` and its variants. The filter function is called on each table entry, checks if this particular entry matches, and if yes, returns the values of the variables bound by the patterns. In its simple form, a filter function has the following type:

```
  #stt: state_type →
  rt: Type0 → (* rt for return type *)
  #tn: table_name (tt_of stt) →
  table_entry #(tt_of stt) tn
→ option rt
```

In this simple form, the filter function does not have access to the state. However, CryptoVerif's semantics allows terms in `get` patterns and conditions to access state: It is possible to have random sampling or other `get` queries inside `get` conditions. For random sampling, read and write access to entropy is needed, and for `get`, it is read access to tables (and get might also need read and write access to entropy). [14] Therefore, we need a full form for filter functions that takes state as parameter and returns a new state alongside the result option type. The reason for the separation in a simple and full form is simplification of the generated F* code for the majority of cases: Most CryptoVerif models will not do random sampling or other `get` queries inside the conditions of `get`.

[14] The terminology *read and write access* might be confusing here, as state is not a global variable to which functions could write. In our case, a function having write access to entropy or state means that it returns a new entropy or state value that must be used subsequently by the caller.

We formalize these two types of filters as follows. We have a type to enumerate the two possibilities

```
type table_filter_type =
  | TableFilterSimple
  | TableFilterFull
```

Then, we have a type `table_filter` that assigns each type the interface of the appropriate filter function. The interesting lines in the following listing are Line 9 and Lines 11 and 12. They define that the simple case only takes a table entry and returns a result, and the full case additionally passes state through.

```
let table_filter
  (#stt: state_type)
  (tft: table_filter_type)
  (rt: Type0)
  (tn: table_name (tt_of stt))
=
  match tft with
  | TableFilterSimple →
      (table_entry #(tt_of stt) tn → option rt)
  | TableFilterFull →
      (st0: state stt → table_entry #(tt_of stt) tn
     → state stt * option rt)
```

Besides `get` and `get[unique]`, we are introducing a third variant for convenience, like cv2ocaml did. It does not exist as such in CryptoVerif, but serves to simplify the generated F* code. In cases where no values retrieved from the table are actually used within the following terms, `get` and `get[unique]` are just a check if matching entries exist. This is simply a boolean result and so we add an `entry_exists` variant in cv2fstar.

The interfaces of the functions implementing the various variants of `get` are similar, so we look at the one for `get` in detail, and discuss only the differences for the others. The non-unique `get` might find multiple matching entries and thus needs read and write access to entropy and thus the state, independently of if the filter function needs write access to state. Thus, we provide only one implementation for the non-unique `get` that always returns a new state. It is parametrized by a table filter type to accomodate for the simple and the full case:

```
1 val get:
2   #stt: state_type →
3   #tft: table_filter_type →
4   #rt: Type0 →
5   st0: state stt →
6   tn: table_name (tt_of stt) →
7   table_filter #stt tft rt tn
8  → state stt * option rt
```

The unique variant of `get` does *not always* need write access to the state, only the full variant does. [15] Thus, we have a function `get_unique_full` that differs from `get` only in that it does not take a table filter type as implicit parameter (so we omit Line 3) and that the table filter is always of type `TableFilterFull` (in Line 7). A `get[unique]` query with a simple filter does not use randomness, and thus does not need entropy: for the function `get_unique`, the table filter is hard-coded to a `TableFilterSimple` one, and the return type is only an `option rt`; no new state is returned. For the existence variant of `get`, we provide `entry_exists_full` and `entry_exists` that differ from `get_unique_full` and `get_unique` in that they return `state stt * bool` and `bool`, respectively.

[15] They always need *read* access because they read tables, so all cv2fstar functions implementing `get` variants take state as parameter.

The implementation of `get` iterates recursively through the table entries and accumulates matches, passing the state through the filter function appropriately. If there is more than one match, an almost uniformly random integer in $[0, m)$ is sampled using `gen_nat`, where $m$ is the number of matches, to choose an entry. The unique variant can stop iterating after the first match is found, as it has been proved by CryptoVerif that it is not possible to have multiple matches. The existence variants of `get` can equally stop iterating after the first successful match.

This concludes the description of the handwritten part of the cv2fstar framework concerning tables. The cv2fstar compiler generates code for a model's instantiation of the table name and table entry types. We will discuss this by example in Section 6.15.3.

## 6.10 TRANSLATING EVENTS

Events are not translated by cv2ocaml; this constitutes an entirely new development in cv2fstar. In a CryptoVerif model, an event can be declared

by

```
event e(t_1, ..., t_n).
```

where `e` is the name of the event, and $t_1$ to $t_n$ are the types of its parameters. An event is invoked by a term

```
event e(M_1, ..., M_n); M
```

where `e` is the name of an event that has been declared in the model, and $M_1$ to $M_n$ are terms with types matching the event's declaration. An event does not fail, so there is no branching, and the return value of the term is the evaluation of the following term $M$.

In Section 6.8, we introduced the `state_type`, containing the `event_type`, and defined that the state contains a list of values of this type. The framework provides a function for appending an event to the event list: The function `state_add_event` takes the state and an event as parameter, and returns a new state with this event added to the list of events, using the `state_upd_events` function.

The cv2fstar compiler generates a model-specific events type, which will be described in Section 6.15.6. Events are important in correspondence queries. cv2fstar is not yet translating these queries, this is left for future work.

## 6.11 Translating Terms

In the previous sections, we discussed single building blocks and their translation: types, randomness, tables, and events. The next step, in this section, are terms. Terms are a building block on our way to translate processes (oracles). However, on the way, translating terms is already enough to translate letfuns. In CryptoVerif, a letfun is nothing else than a term separated into a named function. During a proof, CryptoVerif inlines letfuns immediately for the first game transformation such that they will not be visible anymore as separate functions in any game. For cv2fstar, we decided to keep them separate as a way to structure the code and to avoid code duplication. cv2fstar generates an interface $\mathbb{M}$`.Letfuns.fsti` and an implementation $\mathbb{M}$`.Letfuns.fst` for a model $\mathbb{M}$'s letfuns. We discuss specifics at the end of this section. Before, we describe several aspects of the translation of terms.

A major difference to cv2ocaml is that cv2fstar uses a state-passing style to keep track of the execution state. This complicates the translation of terms, because each term possibly becomes a pair: it has *two* important return values instead of only one; the actual return value, *and* a new state. Although, not all terms in CryptoVerif touch the state, so some optimization is possible to keep the generated F* code readable. Indeed, in our implementation of the cv2fstar compiler inside CryptoVerif, we generate code for `(state, value)` tuples only if a term needs state. However, in this section, to keep the presentation simple, we explain the translation of terms as if all terms needed state and would possibly change it. This means that in the following, all terms produce a new state.

We denote by $\mathbb{G}_{\mathsf{M}}(M)$ the function that produces the F* code translating a CryptoVerif term $M$. In the following, we define $\mathbb{G}_{\mathsf{M}}(M)$ step by step for the different kinds of terms.

VARIABLE NAMES, $\mathbb{G}_{\mathsf{M}}(x)$. A variable term $x$ is translated as follows:

```
𝔾_M(x) =
  state, 𝔾_var(x)
```

where $\mathbb{G}_{\mathsf{var}}(x)$ is a function taking a CryptoVerif variable name and returning an F* variable name. We use a prefix `var_` and an encoding such that this function is injective, to avoid collisions with other functions that generate names.

CONSTANTS. Constants are translated to their annotated name:

```
𝔾_M(c) =
  state, 𝔾_const(c)
```

At first sight, it might appear wrong that variables and constants are translated as pairs. However, in the following, it will become clear that before using any term, the pair of `state` and value is always decomposed.

TUPLES. In CryptoVerif, tuples are constructed by wrapping multiple terms into parentheses, separated by commas, like in this example: `(a, b, c)`. Tuples are of type `bitstring`. When translating a tuple, we feed each element through the serialization function of its type and concatenate the resulting byte sequences. Each element of the tuple might process state, so we first chain the preparation of each individual element. For terms $M_i$ of types $t_i, 1 \leq i \leq n$, we generate the following code:

```
𝔾_M((M_1,...,M_n)) =
  let state, v_1 = 𝔾_M(M_1) in
  ...
  let state, v_n = 𝔾_M(M_n) in
  state, compos [𝔾_ser(t_1) v_1; ...; 𝔾_ser(t_n) v_n]
```

The function compos takes a list of byte sequences as input. It uses a 4-byte prefix to encode the number of tuple elements, and a 4-byte prefix in front of each element to encode its length. This makes tuple composition limited to terms that result in a serialization with a maximum length of $2^{32}-1$ bytes. For longer strings, the tuple composition function will instead return an error symbol. This is another practical limitation of cv2fstar that could be pushed further if need be.

FUNCTION CALLS WITHOUT ENTROPY. Function calls are generated with the function name from the implementation annotations. As previously, each parameter is translated using $\mathbb{G}_{\mathsf{M}}$. For a function $f$, and terms $M_i, 1 \leq i \leq n$, we generate:

```
𝔾_M(f(M_1,...,M_n)) =
  let state, v_1 = 𝔾_M(M_1) in
  ...
  let state, v_n = 𝔾_M(M_n) in
  state, 𝔾_f(f) v_1 ... v_n
```

FUNCTION CALLS WITH ENTROPY. If a function $f$ needs entropy because it internally samples random values, we wrap it with `call_with_entropy`. This function is already of the form that it returns a tuple of state and a value, so we do not need an explicit `state` in the last line.

```
𝔾M(f(M1,...,Mn)) =
  let state, v1 = 𝔾M(M1) in
  ...
  let state, vn = 𝔾M(Mn) in
  call_with_entropy state (𝔾f(f) v1 ... vn)
```

For simplicity, we use the same variable names $v_1$ to $v_n$ in this and the other listings; in the actual implementation of the compiler, we use unique names. The function $f$ is required to take entropy as last parameter such that ($\mathbb{G}_f(f)$ $v_1$ ... $v_n$) results in a function with only entropy as parameter.

In Section 6.4, we mentioned that it is possible to provide an annotation for a CryptoVerif letfun such that it is implemented by a function, hiding the CryptoVerif body of the letfun. In this case, the letfun is treated as a non-letfun function and translated as described here, as a function call with or without entropy. The following paragraph describes letfun function calls where the letfun is translated with its body as it is.

LETFUN FUNCTION CALLS. Letfun calls are translated to a function call of the appropriate function in the `𝕄.Letfun` module. The generation of this module is described at the end of this section. We are again assuming that each letfun processes state and returns a pair. In the actual implementation, we optimise the case in which a letfun does not process state. For a letfun call that has the terms $M_i, 1 \leq i \leq n$ as parameters, we generate:

```
𝔾M(letfun(M1,...,Mn)) =
  let state, v1 = 𝔾M(M1) in
  ...
  let state, vn = 𝔾M(Mn) in
  𝕄.Letfun.fun_letfun state v1 ... vn
```

The F* function name of a letfun is its CryptoVerif name prefixed with `fun_`.

EQUALITY TEST. For an equality test $\mathtt{M}_1 = \mathtt{M}_2$, or its negation $\mathtt{M}_1 <> \mathtt{M}_2$, CryptoVerif already ensures that $M_1$ and $M_2$ have the same type $t$. An equality test is translated to a function call as indicated by the implementation annotation of the type $t$:

```
𝔾M(M1 = M2) =
  let state, v1 = 𝔾M(M1) in
  let state, v2 = 𝔾M(M2) in
  state, 𝔾eq(t) v1 v2
```

The syntax <> from CryptoVerif is translated to the negation of the equality test function of type $t$:

```
𝔾M(M1 <> M2) =
  let state, v1 = 𝔾M(M1) in
  let state, v2 = 𝔾M(M2) in
  state, not (𝔾eq(t1) v1 v2)
```

IF CONDITIONALS. The translation of an if condition is as follows, for a boolean term $M$, and branches $M'$ and $M''$:

```
𝔾M(if M then M' else M'') =
  let state, v = 𝔾M(M) in
  if v then 𝔾M(M') else 𝔾M(M'')
```

RANDOM SAMPLING. For a type $t$ that allows random sampling, and a following term $M$, we generate the following code:

```
1 𝔾M(x <-R t; M) =
2  let state, 𝔾var(x) = call_with_entropy state 𝔾$(t) in
3  𝔾M(M)
```

EVENTS. For an event $e$ with parameters $M_i, 1 \leq i \leq n$ and a following term $M$, we generate the following code:

```
1 𝔾M(event  e(M1,...,Mn); M) =
2  let state, v1 = 𝔾M'(M1) in
3  ...
4  let state, vn = 𝔾M'(Mn) in
5  let ev = Event_e v1 ... vn in
6  let state = state_add_event state ev in
7  𝔾M(M)
```

Here, $\mathbb{G}_{M'}$ is a slightly adapted version of $\mathbb{G}_M$ that translates replication indices if they occur in a particular way in the parameters of the event: if the oracle in which the event occurs is under multiple nested replications, the replication indices of all levels must be used within a tuple in the right order, from the innermost to the outermost replication index; if the oracle is under only one replication, the replication index must be used directly; in both cases, this has to be directly as a parameter of the event, not nested inside other terms inside a parameter, e. g., function calls. For example, for an event inside of two nested replications, the following event invocation is supported by cv2fstar:

```
1 foreach i1 ≤ Q1 do
2  ...
3  foreach i2 ≤ Q2 do
4   ...
5   event e1((i2, i1), ...);
6   ...
```

The first parameter of the event `e1` has to be declared as type bitstring. For event `e2` in the following example, inside of only one replication, the type of `e2`'s first parameter must be the replication parameter, in this case `Q1`.

```
1 foreach i1 ≤ Q1 do
2  ...
3  event e2(i1, ...);
4  ...
```

If these conditions are met, we translate a tuple of replication indices as `(serialize_nat sid)` because the event's parameter expects a `bytes` value, and a single replication index as `sid` because the event's parameter expects a `nat` value. As we describe in detail in the next section, the variable `sid` of type `nat` always contains a current session ID that reflects all replications.

LET BINDINGS. The most interesting translation might be the one of let bindings, because it includes pattern matching. A pattern matching in CryptoVerif generally has the following form:

```
1 let PAT = M in M' else M''
```

where `PAT` is the pattern, $M$ is the term that is being matched to the pattern, $M'$ is the term evaluated in case of success of the pattern matching, and $M''$ is the term evaluated otherwise.

Pattern matching can be split into three cases for PAT: a single variable assignment, a single equality pattern, and finally everything more complex than that. We go through them one by one.

A single variable assignment is translated as such to F*, it always succeeds and thus, it has no else branch $M''$:

```
1 𝔾M(let x = M in M′) =
2   let state, v = 𝔾M(M) in
3   let 𝔾var(x) = v in
4   𝔾M(M′)
```

A single equality pattern is translated to an if condition in F*. The generic case looks like this, with a term $M_1$ of type $t_1$:

```
1 let (=M1) = M in M′ else M″
```

The condition in F* uses the equality test function of this type. The semantics of CryptoVerif defines that $M$ is evaluated first, and then $M_1$.

```
1 𝔾M(let (= M1) = M in M′ else M″) =
2   let state, v = 𝔾M(M) in
3   let state, w = 𝔾M(M1) in
4   if 𝔾eq(t1) v w then 𝔾M(M′) else 𝔾M(M″)
```

Finally, we look at a more complex pattern with one or more variable assignments and equality patterns, and one ore more function inversions or tuple decompositions, possibly nested. The generic algorithm for this is quite involved, so we describe it only on a high level, and based on an example that contains all the kinds of patterns listed above.

On a high level, when translating a complex pattern, we go through it inwards, starting from the outermost layer. We collect variable assignments, equality patterns, and inversions (function inversions or tuple decomposition). Then, we first generate code for the inversions. Generally, for all failure cases, we generate the code for the else case of the pattern matching, $M''$. After the success case of an inversion, we pass through the variable names of assignments at that layer and continue with the next inversion. After all inversions are done, we translate the equality patterns as an if condition, combining multiple ones with &&. The success case continues with the translation of $M'$, the else case again with $M''$.

Let us look at the example that we are using to showcase the translation algorithm:

```
1 let (=M1, b: t2, f(c: t4, =M5)) = M in M′ else M″
```

On the outermost layer, the pattern is a tuple of three elements. The first one is an equality pattern, and we suppose that the term $M_1$ has type $t_1$. The second one is an assignment to a variable $b$ of type $t_2$. The third one is an application of the function $f$, which must have been declared `[data]` to have an efficiently computable inverse $f^{-1}$. We suppose that $f$ returns a value of type $t_3$. Inside of the function $f$, we have an assignment to a variable $c$ of type $t_4$, and an equality pattern with term $M_5$ of supposed type $t_5$.

Moving on to the code that cv2fstar generates for this example. First, the terms $M$, $M_1$ and $M_5$ need to be processed in the right order. The semantics of CryptoVerif defines that $M$ is evaluated first, and then the terms of the equality patterns $M_1$ and $M_5$:

Listing 6.3: F* code generated by cv2fstar for the pattern matching example.

```
1 let state, v = 𝔾_M(M) in
2 let state, v_1 = 𝔾_M(M_1) in
3 let state, v_5 = 𝔾_M(M_5) in
```

The next step is the decomposition of the tuple. We use a function `decompos` of our framework that takes a byte sequence and decomposes it into its parts according to the format used by `compos`: a first 4-byte prefix encodes the number of parts, and each part is prefixed by 4 bytes encoding its length. If there is an inconsistency with the prefixes, `decompos` returns `None` (Line 9 in Listing 6.4), otherwise it returns a list `Some` $[c_1, \ldots, c_n]$, where $c_1, \ldots, c_n$ are the byte sequences representing the tuple's elements (Line 5). We then apply the deserialization functions of the appropriate types

Listing 6.4: Continuation: F* code generated by cv2fstar for the pattern matching example.

```
4 match (λ n_1 → match decompos n_1 with
5               | Some [c_1; c_2; c_3] →
6                  (match (𝔾_deser(t_1) c_1, 𝔾_deser(t_2) c_2, 𝔾_deser(t_3) c_3) with
7                  | (Some r_1, Some r_2, Some r_3) → Some (r_1, r_2, r_3)
8                  | _ → None)
9               | _ → None) v with
10 | None → 𝔾_M(M'')
11 | Some (r_1, 𝔾_var(b), r_3) →
```

to these byte sequences (Line 6). If all deserializations succeed, we return the values; otherwise we return `None` (Lines 7 and 8). For the case that the decomposition of the tuple failed overall (`None` in Line 10), we generate the code for term $M''$. If it succeeds, we carry on with the next patterns, and pass on a variable for each element of the tuple that was just decomposed (Line 11). As the second tuple element is a variable assignment, we use the variable's name $\mathbb{G}_{\text{var}}(b)$. The $\lambda$ function starting in Line 4 in this code listing corresponds to the inversion function $\mathbb{G}_{\text{inv}}$ of the tuple.

Now, it is the turn of the inversion of the function $f$. We use the function name of the inverse function $\mathbb{G}_{\text{inv}}(f)$ as indicated by the implementation annotation. It takes the third tuple element $r_3$ as parameter. For the case that inversion fails, we again translate the code of $M''$. For the case that it succeeds, we pass along variable $c$ and and the second matched parameter $r_5$:

Listing 6.5: Continuation: F* code generated by cv2fstar for the pattern matching example.

```
12   match 𝔾_inv(f) r_3 with
13   | None → 𝔾_M(M'')
14   | Some (𝔾_var(c), r_5) →
```

Finally, we produce the code for the equality pattern's tests. We use the equality test functions of the appropriate types to compare the matched elements $r_1$, $r_5$ with the terms $v_1$, $v_5$ that have been evaluated at the beginning. For the case that the tests succeed, we translate the code for $M'$, otherwise again the code for $M''$:

Listing 6.6: End: F* code generated by cv2fstar for the pattern matching example.

```
      if 𝔾eq(t1) r1 v1 && 𝔾eq(t5) r5 v5
      then 𝔾M(M′)
      else 𝔾M(M″)
```

In this example throughout the last 4 listings, it is clear that the term $M''$ appears multiple times. If $M''$ translates to multiple lines of code, the repetition can make the generated code quite unreadable. We optimise our compiler for the case of a multiline $M''$ and wrap the pattern matching in an additional `match`. While translating the original pattern matching, we replace $M''$ by `None` and $M'$ by the list of variable's assigned. For our example, this results in the following:

```
let state, v = 𝔾M(M) in
let state, v1 = 𝔾M(M1) in
let state, v5 = 𝔾M(M5) in

let r = (* the remainder of the above pattern matching code,
        with "𝔾M(M′)" replaced by "Some (𝔾var(b), 𝔾var(c))"
        and "𝔾M(M″)" replaced by "None" *) in

match r with
| None → 𝔾M(M″)
| Some (𝔾var(b), 𝔾var(c)) → 𝔾M(M′)
```

In the last two lines, $\mathbb{G}_{\mathsf{M}}(M')$ and $\mathbb{G}_{\mathsf{M}}(M'')$ are again the original ones. In this style, $\mathbb{G}_{\mathsf{M}}(M'')$ is guaranteed to be translated only once.

TABLE INSERTS. For a table $\mathrm{tbl}$, an insert with terms $M_i, 1 \leq i \leq n$ as parameters, and a following term $M$, we generate the following code:

```
𝔾M(insert tbl(M1,...,Mn);M) =
  let state, v1 = 𝔾M(M1) in
  ...
  let state, vn = 𝔾M(Mn) in
  let entry = TableEntry_𝔾tbl(tbl) #Table_𝔾tbl(tbl) v1 ... vn
  let state = insert state Table_𝔾tbl(tbl) entry in
  𝔾M(M)
```

In the construction of the table entry we need to indicate the table name as implicit parameter with # because F* cannot infer it, in general, from the parameters.

TABLE GET QUERIES. We remind the generic syntax of a get query:

```
get tbl(p1, ..., pn) suchthat M then M′ else M″
```

The patterns $\mathsf{p}_i, 1 \leq i \leq n$ can consist of variable assignments, equality tests, tuple decompositions, and function inversions, just like the patterns in let bindings. The term $M$ constitutes an additional boolean condition.

We are now describing how we generate the filter function that was first introduced in Section 6.9. We call it $\mathbb{G}_{\mathsf{filter}}$ in the following and the generated code is shown in Listing 6.7. It is passed as a parameter to the cv2fstar functions `get`, `get_unique(_full)`, and `entry_exists(_full)`. The filter function takes two parameters, the state of the model-specific type `m_state`, and a table entry of type `m_table_entry` parametrized by

the table name Table_$\mathbb{G}_{\text{tbl}}(\text{tbl})$. After decomposing the entry into its fields

Listing 6.7: F* code for $\mathbb{G}_{\text{filter}}$ generated by cv2fstar.

```
(λ (state:m_state) (te:m_table_entry Table_G_tbl(tbl)) →
 let TableEntry_tbl v_1 ... v_n = te in
 (* translate the patterns using the complex case from above,
    use "None" as M'' and the following as M': *)
 let state, v = G_M(M) in
 if v then state, Some (u_1, ..., u_m) else state, None
)
```

$v_i, 1 \leq i \leq n$, we are matching these $n$ values to the $n$ patterns $p_i$. For this pattern matching, we reuse the pattern matching algorithm of let bindings. The difference is that we have $n$ patterns to match to $n$ values, and not only a single one. We adapt to this case by chaining the $n$ pattern matchings one after another. For the $n$-th pattern matching, we use special values for the success and failure cases: `None` for the failure case, and the code for the `suchthat` condition in the success case, see Lines 5 and 6. The $u_i, 1 \leq i \leq m$ are the variables that appear in variable assignments in the patterns *and* that are used in $M'$. These are the variables that are returned as answer of the get query.

The filter function $\mathbb{G}_{\text{filter}}$ is used as follows in the translation of get queries. For a non-unique get query with a success term $M'$ that uses variables returned from the query, we generate the following code:

```
(* evaluation of terms appearing in equality patterns *)
let state, v... = G_M(...) in

match get state Table_tbl G_filter with
| state, None → G_M(M'')
| state, Some (u_1, ..., u_m) → G_M(M')
```

First, the terms of any equality patterns used in the query need to be evaluated before anything else. Then, we call the function `get` and pass it the state, the table name, and the filter function. Finally, we evaluate the next terms $M'$ and $M''$ according to the query result.

For a `get[unique]` query, the translation is the same except that the function name `get` is replaced by `get_unique_full`. For a get query with a success term $M'$ that does not use any variable returned from the query, we call `entry_exists_full` instead. It returns a boolean result, based on which we decide if $M'$ or $M''$ is evaluated:

```
(* evaluation of terms appearing in equality patterns *)
let state, v... = G_M(...) in

let state, b = entry_exists_full state Table_tbl G_filter
if b then G_M(M') else G_M(M'')
```

TRANSLATING LETFUNS. Having seen how terms are translated, we can come back to letfuns and look at how their interfaces and implementations are generated.

For the interface, we generate a `val` declaration in the file `M.Letfun.fsti` for each letfun. As name we use the same name as in the CryptoVerif model,

prefixed with `fun_`. If the letfun's term needs state, the first parameter of the letfun is state. If the letfun has parameters, their types are appended separated by $\rightarrow$. Finally, for the return types, if the letfun's term needs state, the letfun's return type is $\underline{m}$`_state * a`, where `a` is the original return type, and only `a` if the term does not need state. Assuming a letfun that needs state and takes $n$ parameters of type $t_i, 1 \leq i \leq n$, and returns a value of type $t$, the declaration is as follows:

```
val fun_letfun : m_state → G_t(t_1) → ... → G_t(t_n) → m_state * G_t(t)
```

For the implementation, we generate a `let` in the file $\mathbb{M}$`.Letfun.fst` for each letfun. We use the same variable names for the parameters as in the CryptoVerif model. If the letfun's term needs state, the first parameter is `state`. For the body we call into $\mathbb{G}_{\mathsf{M}}$. Assuming a letfun that needs state, takes $n$ parameters $p_i, 1 \leq i \leq n$, and has the body $M$, the generated code is as follows:

```
let fun_letfun state G_var(p_1) ... G_var(p_n) =
  G_M(M)
```

## 6.12 Translating Oracles as Functions

Overview. Oracles, or processes in the channels frontend, are the essential top-level building block of CryptoVerif models. We use the terminology oracles and processes interchangeably, but prefer oracles because this is how they are called in cryptographic proofs. Oracles are identified by a name, can have parameters, and in our context, can end either with a return statement, or a yield statement returning control to the caller without returning values[16]. Oracles are translated to F$^\star$ functions, grouped by the module block annotation in which they are contained: The interfaces of a module's oracles are written to the file $\mathbb{M}$`.Module.fsti` and their implementation to the file $\mathbb{M}$`.Module.fst`, where `Module` is the module name used in the module block annotation.[17] Reusing the translation of terms, we only need to add treatment of `yield` and `return` to translate an oracle's implementation. Oracles can be part of a series of subsequent oracles. We enforce their order and implement the sharing of variables between them through session state identified by a session ID.

[16] A third possibility is `event_abort`, which ends execution of the entire system. We do not allow this in cv2fstar's restricted input language. See Section 6.4.2 for details.

[17] As a reminder, $\mathbb{M}$ is the name of the CryptoVerif model, derived from its filename, converted to all-uppercase letters.

### 6.12.1 *Session Data Structure and Session IDs*

Motivation. An oracle following after one or more oracles is allowed to use variables of the preceeding oracles. Otherwise said, such a following oracle can have free variables that are neither defined by the oracle's input parameters nor within the oracle itself, but have been defined by a previous oracle or its inputs. In cv2fstar, oracles are translated to separate functions and thus, cannot directly share variables. We use a session data structure to overcome this. At the end of an oracle's F$^\star$ code, all free variables of following oracles are written into a session entry. At the beginning of a following oracle, these variables are read from the session entry. A session entry is identified by the name of the following oracle, the counter of the return statement within the current oracle, and a session ID. While the

attribution of session IDs is discussed in the next paragraph, we explain the return counter. An oracle can have multiple return statements, due to the fact that branching is possible through different means (`if`, `get`, and `let` binding with pattern matching), and the fact that CryptoVerif's syntax does not allow to re-merge branches.[18] This means oracle definitions need to be repeated after each return. We enforce that the structure of oracles is the same after each return: The oracle's names, whether they are replicated, and their input and output types, must be the same after all returns. However, the body of an oracle can be different in each definition. Thus, it can have a different set of free variables that need to be stored in the session entry by the previous oracle.

[18] For example, the following is *not* allowed to continue with a term $M'$ in both branches after an if condition:
`(if` $M$ `then` $M_1$ `else` $M_2$`);` $M'$

The reason for this restriction of the oracle structure is the following. Assume we allow for different following oracles after different returns of an oracle $O$. In general, the adversary does not know which branch and thus which return was taken in $O$. Thus, the adversary would not know if an oracle that it wants to call after $O$ is actually available. For example, if in the first branch, $O_1$ is available as a following oracle, and $O_2$ in the second branch; if the first branch was taken and the adversary calls $O_2$, this oracle is not available. How do we encode this error? It should be different from a `yield`, because that is used to encode that an *existing* oracle did not return. The restriction to the same oracle structure has been made to avoid the need of extending the semantics of CryptoVerif to encodes this type of error. As a side effect, this means that the adversary cannot detect which branch was taken within an oracle only based on the metadata of the following oracles (name, replication, input and output types).[19]

[19] Information about the branch taken can be encoded into the return *values*.

We use the following example to illustrate the restrictions:

```
1  O1(...) :=                          11
2   if ... then (                      12   ) else (
3     return (...);                    13     return (...);
4     O2(...) :=                       14     O2(...) :=
5       ...                            15       ...
6       return (...);                  16       return (...);
7     O3(...) :=                       17     O3(...) :=
8       ...                            18       ...
9       return (...)                   19       return (...)
10                                     20   )
```

In this example, the two columns correspond to the `then` branch (Lines 3 to 9) and the `else` branch (Lines 13 to 19) of the `if` condition. Returns at the same place in both branches must have the same number of parameters, and the same types (Lines 3 and 13, 6 and 16, 9 and 19). Oracle definitions at the same place in both branches must have the same name, the same number of input parameters, and the parameters must be of the same types (Oracles `O2` and `O3`). The bodies of Oracles `O2` and `O3` are allowed to be different in both branches (parts omitted by `...` on Lines 5 and 15, and 8 and 18).

ORACLE NAME, SESSION ENTRY, SESSION ID. A session represents which oracles can be called at that stage of execution. Each session is identified by a session ID. Due to replication and parallel composition, multiple oracles

might be callable from a given session. We store a list of session entries associated to each session ID in a map. If an oracle has multiple following oracles in parallel composition, the oracle adds a session entry for each of them. Session entries for oracles under replication are never removed.

We use a type $\underline{m}$_oracle_name to enumerate all oracles:

```
type m_oracle_name: eqtype =
  | Oracle_O_1
  ...
  | Oracle_O_n
```

Here, the $O_i, 1 \leq i \leq n$ are the names of the oracles in the CryptoVerif model, over all modules. The value `Oracle_`$O_i$ is used to regroup all session entries valid for this oracle, as we will see later.

Session entries look like this:

```
noeq type m_session_entry =
  | ...
  | Rj_O_i: v_1:t_1 → ... → v_m:t_m → m_session_entry
  | ...
```

Here, $j$ stands for the counter of the return in the previous oracle, and $O_i$ is the oracle that can be called. The values $\mathtt{v}_k$ of type $\mathtt{t}_k, 1 \leq k \leq m$ are the free variables of oracle $O_i$ or one of its following oracles. The oracle will load these variables from the session entry.

### 6.12.2 *The Interface of an Oracle Function*

For an oracle $O$ with input types $t_i, 1 \leq i \leq n$, and output types $t'_j, 1 \leq j \leq m$, we generate the following interface:

```
val oracle_O : m_state → nat → G_t(t_1) → ... → G_t(t_n)
  → m_state * option (nat * G_t(t'_1) * ... * G_t(t'_m))
```

The input session ID of type `nat` is only present if the oracle is *not* a top-level oracle: top-level oracles are the entry points to a model, and so there is no session to continue, yet. Non-top-level oracles always continue a session and thus, *always* take a session ID as input. If the oracle takes no input parameters, no types $\mathbb{G}_\mathsf{t}(t_i)$ are indicated in the interface except $\underline{m}$`_state`.

In general, an oracle's function returns `state, None` when the oracle terminates by `yield`, and it returns `state, Some(sid,` $v_1$`, ...,` $v_m$`)` when it terminates by `return(`$M_1$`, ...,` $M_m$`)`. In more detail, the behaviour is as follows: A new session is started by oracles under replication that have following oracles. Only in this case, a new session ID of type `nat` is returned along with the oracle's other return values. If the oracle has no output, i. e., the return statements are empty, the output types are replaced by a single `unit`. If the oracle does never return in any branch, but always finishes by `yield`, the return type of the function is only $\underline{m}$`_state`. Such an oracle cannot have following oracles, so there is never a new session ID.

### 6.12.3 *The Implementation of an Oracle Function*

We start by discussing the implementation of top-level oracle functions. They are less involved, because there is no need to resume a session. Just like we defined a function $\mathbb{G}_\mathsf{M}(M)$ to translate a term $M$, we define a function

$\mathbb{G}_{\mathsf{O}}(P)$ to translate oracle bodies $P$. It reuses $\mathbb{G}_{\mathsf{M}}$ for everything except `yield` and `return`. We define $\mathbb{G}_{\mathsf{O}}$ for them in the course of this section.

For an oracle $O$ with input parameters $v_i, 1 \leq i \leq n$ of types $t_i, 1 \leq i \leq n$, the implementation starts as follows:

```
let oracle_O state G_var(v_1) ... G_var(v_n) =
```

There is no session ID parameter.

If there are following oracles after this oracle, we need to write one or more session entries before returning. Thus, we need a session ID to which these entries are associated. If the current oracle is under replication, which is always the case for top-level oracles, we need a new session ID. It is obtained by the following function call that reserves a new session ID in the session map:

```
  let state, sid = state_reserve_session_id state in
```

If the current oracle is not under replication, the same session ID is reused. This is important for non-top-level oracles. The reason why we reserve a session ID so early within an oracle function is that we might need it for an event that uses it in its parameters. This is not optimal, as we might "lose" a session ID in cases where there is no such event and the oracle ends by `yield`. This is a tradeoff to keep the code uniform and simple. The alternative of only reserving the session ID right before it is needed, and thus possibly in multiple branches of the oracle, is more complex.

CryptoVerif allows pattern matching in oracle inputs:

```
O(p_1, ..., p_n) := ...
```

The depth of the patterns[20] does not influence the type of the input parameters: this oracle has $n$ input parameters, and their types are the types of the patterns $\mathtt{p}_i, 1 \leq i \leq n$. The pattern matching is evaluated inside the oracle body before the remaining code of the oracle, to bind the variables and check the equality patterns. If the pattern matching fails, the call to the oracle counts regardlessly and the oracle ends with `yield`.[21] We generate the appropriate code by using the same algorithm as for the complex case of let binding pattern matching. For the success case, we generate code for the body of the oracle using $\mathbb{G}_{\mathsf{O}}$. For the failure case, we use $\mathbb{G}_{\mathsf{O}}(\mathtt{yield})$.

[20] Each pattern might contain multiple levels of function inversions.

[21] So, pattern matching in oracle parameters is just syntactic sugar for applying the pattern to the input parameters as first step inside the oracle body.

TRANSLATING YIELD. If all branches within an oracle finish by `yield`, the return type of the oracle is (only) $\underline{m}$`_state`. Thus, we translate `yield` by just `state`. If not all branches finish by `yield`, we translate it by `state, None`. This is the definition of $\mathbb{G}_{\mathsf{O}}(\mathtt{yield})$.

TRANSLATING RETURN. In a simple case where there are no following oracles after a return, we can translate it as follows, considering a return term $\mathtt{return}(M_1, \ldots, M_n)$:

```
  let state, v_1 = G_M(M_1) in
  ...
  let state, v_n = G_M(M_n) in
  state, Some (v_1, ..., v_n)
```

If the oracle has following oracles, we need to create session entries and return a session ID. We need one session entry for each following oracle, to provide it with the values of its free variables (and those of its own following

oracles, possibly). As described above, the definition of the following oracle and its free variables can depend on the branch taken in the current oracle. Thus, it makes sense to create the session entries right before returning.

In the following code, we translate return number $r$ of the current oracle, returning a tuple of terms $M_i, 1 \leq i \leq n$. We assume there are $m$ following oracles $O_j, 1 \leq j \leq m$ that have $n_j$ free variables each, called $b_{j,k}, 1 \leq k \leq n_j$.

```
𝔾O(return(M_1,...,M_n)) =
 let sel = [Rr_O_1 𝔾var(b_{1,1}) ... 𝔾var(b_{1,n_1});
            ...;
            Rr_O_m 𝔾var(b_{m,1}) ... 𝔾var(b_{m,n_m})] in
 let state = state_add_to_session state sid sel in
 let state, v_1 = 𝔾M(M_1) in
 ...
 let state, v_n = 𝔾M(M_n) in
 state, Some (sid, v_1, ..., v_n)
```

The function `state_add_to_session` adds the list of session entries `sel` to the already existing session entries of session `sid`. If it is a new session, these will be the only entries.

NON-TOP-LEVEL ORACLES. We continue with the implementation of non-top-level oracle functions:

```
let oracle_O state sid 𝔾var(v_1) ... 𝔾var(v_n) =
```

We always need the session ID parameter `sid` for non-top-level oracles.

Non-top-level oracles start by retrieving an appropriate session entry from the list of session entries associated with the session ID `sid`.

```
match get_session_entry state Oracle_O sid with
| None → 𝔾O(yield)
| Some (se: m_session_entry) →
```

The session might contain session entries for different oracles, but there can only be one that fits to oracle $O$. The reason for this is that within *one* session, only one return of the previous oracle could have been taken. Taking multiple different return branches is only possible if the previous oracle is under replication, and then the session entries end up in lists for different session IDs. We pass the oracle name to the session retrieval function such that it finds the at most one session entry that fits. The function `get_session_entry` returns `None` in case the session ID does not exist or there is no fitting session entry. In this case, the oracle function fails early. The function `get_session_entry` is used in oracles under replication, as it does not remove the session entry. For oracles not under replication, we use `get_and_remove_session_entry` that removes the session entry. It returns a new state alongside the result.

After retrieving the session entry, we continue with the code for reserving a new session ID if needed, as explained above in the paragraph about top-level oracles. Then, we generate code to match on the different possibilities for the session entry. There is one possibility for each return of the previous oracle $O_p$. Considering a previous oracle with $m$ returns, and $n_j$ variables transmitted from $O_p$ to $O$ at return number $j$, we generate the following code:

```
match se with
| R1_O_p 𝔾var(u_{1,1}) ... 𝔾var(u_{1,n_1}) →
   (* code for body of oracle O after return 1 of O_p *)
...
| Rm_O_p 𝔾var(u_{m,1}) ... 𝔾var(u_{m,n_m}) →
   (* code for body of oracle O after return m of O_p *)
```

The code within the branches is generated as in top-level oracles: we match the input patterns of the oracle and continue with $\mathbb{G}_{\mathsf{O}}(P)$ in the success case, where $P$ is the oracle's body, and continue with $\mathbb{G}_{\mathsf{O}}(\texttt{yield})$ in the failure case.

### 6.12.4 *Session Type*

Before we end the section on the translation of oracles, we give a quick overview of the declaration and implementation of the session type. Like for tables, we define a session type that holds information specific to a CryptoVerif model. It has a single constructor:

```
noeq type session_type =
  | SessionType :
     (* oracle name *)
     on: eqtype →
     (* session entry *)
     se: Type0 →
     (* following oracles *)
     fo: (on → list on) →
     (* session_entry_to_name *)
     se2on: (se → on)
    → session_type
```

Similar as with tables, we have one type enumerating all possible oracle names (`on`), and one type for the actual session entries (`se`). We define getters `oracle_name` and `session_entry` for these types, returning `on` and `se` for a given session type.

The field `fo` must be instantiated by a function mapping an oracle name to the oracle names of all allowed following oracles. The field `se2on` must contain a function mapping session entries to the oracle's name to which they belong. cv2fstar generates the appropriate instantiations for a model. The function `se2on` is generated such that for each session entry `R`$j$`_`$O_k$ it returns the oracle name `Oracle_`$O_k$. It is used by `get_session_entry` and `get_and_remove_session_entry` to determine if a session entry fits to a given oracle name; `se2on` is used to implement an `is_valid_entry` predicate that checks whether an oracle name and a session entry match:

```
let is_valid_entry
  (#st: session_type)
  (on: oracle_name st)
  (se: session_entry st)
=
  match st with | SessionType _ _ _ se2on → (se2on se = on)
```

We plan to use the function `fo` for future work, when we prove lemmas about the correct treatment of sessions by oracle functions (see Section 6.17).

All sessions of a model are held in a type `sessions` that is refined by the session type. It is implemented as a map from a natural number session ID to a list of session entries defined for this session type:

```
let sessions (st: session_type) = Map.t (list (session_entry st))
```

Another example of where `is_valid_entry` is used, is the return type of `get_session_entry` and `get_and_remove_session_entry`:

```
val get_and_remove_session_entry
  (#stt: state_type)
  (st: state stt)
  (on: oracle_name (st_of stt))
  (sid: nat)
: (state stt
   * option (s: (session_entry (st_of stt)){is_valid_entry sn s}))
```

This refinement is useful for the implementation of the oracle body. When matching on the different possibilities for the session entry, the refinement allows us to only match on the session entries that are actually possible for the oracle's name: The F* typechecker understands that the other session entries are not possible and does not complain about a non-exhaustive pattern matching.

## 6.13 Translating Assumptions as Lemmas

Many assumptions are made in a CryptoVerif model: On the one hand, there are cryptographic hypotheses on the cryptographic building blocks. On the other hand, there often are functions for message formats that are assumed to have certain properties, e. g., producing disjoint output for different message types, or having an inverse. Functions for serialization and deserialization are assumed to be inverses, equality test functions are assumed to be equivalence relations.

While we are not hoping to prove the cryptographic assumptions for an implementation in F*, proving that the non-cryptographic assumptions hold for an implementation is definitely in the realm of F*'s type system. With cv2fstar, we translate many of theses assumptions to F* lemmas. This produces proof obligations for the implementation, that the user has to prove. [22] This reinforces the link between CryptoVerif models and F* implementations produced by cv2fstar. For the proofs of the lemmas, existing lemmas in F* and HACL* can of course be reused; so some lemmas might be straightforward to prove. In the following, we discuss what kind of lemmas cv2fstar produces for which kind of assumption in a CryptoVerif model. The generated code is written to the file `M.Equations.fsti`. The user will need to write proofs for the lemmas in the file `M.Equations.fst`.

[22] The user can also consciously admit these lemmas and thus assume they have a proof, but at this point, cv2fstar did all it could by translating the lemmas.

Sources of Lemmas. There are 4 sources of lemmas from the point of view of cv2fstar, at the moment. First, there are equations explicitly defined in the CryptoVerif model. Second, the assumption that the inverses of functions declared with `[data]` are correct. Third, serialization and deserialization functions are assumed to be inverses of each other. Fourth, CryptoVerif allows to declare built-in equational theories for functions, like commutativity, up to forming a group with an inverse function and a neutral element. cv2fstar generates lemmas for these kinds of assumptions. We go through them one by one and explain what lemma code we generate for these. Generally, we produce `val` declarations of lemmas, and no proofs.

### 6.13.1 *Explicitly Defined Equations*

An equation is defined starting with the `equation` keyword, followed by a list of universally quantified variables with type indication, then a boolean term $M$ encoding the assumption, and optionally, the keyword `if` followed by a condition $M'$:

```
equation forall v_1:t_1, ..., v_n:t_n; M if M'.
```

We generate the following F* code for such an equation:[23]

```
val lemma_i: unit → Lemma (∀ (G_var(v_1):G_t(t_1)) ... (G_var(v_n):G_t(t_n)).
  G_M(M') ⇒ G_M(M))
```

[23] Indicating `unit` as a parameter is required in F* even if the lemma takes no actual parameters.

Here, $i$ is the number of the lemma: We do not generate speaking lemma names but just numerate them. The lemma does not take any parameters (`unit`), the universally quantified variables are all defined inside. If $M'$ is left empty, it is implicitly `true`, and we omit $\mathbb{G}_{\mathsf{M}}(M') \Rightarrow$. Equations are not allowed to contain terms that process state. We use the version of $\mathbb{G}_{\mathsf{M}}$ that is optimized for terms not processing state. It does not produce code that returns a pair of `state` and the evaluation of the term, but code that returns only the evaluation of the term.

The cryptographic library shipped with CryptoVerif contains many equation definitions, mostly within the macros that define cryptographic assumptions. If we wanted to translate those, we would need F* translations for all types, functions, and constants used within the macros. However, a user might not want to translate all the cryptographic internals; most likely they just want to use a ready-made implementation from HACL*. So, we translate only those equations that use only types, functions, and constants that have an implementation annotation in the CryptoVerif model. We skip all other equations.

### 6.13.2 *Correctness of Inverses*

For a function $f$ with inverse $f^{-1}$, we generate lemmas for two assumptions: First, $f^{-1}$ applied to the output of $f$ for some input $x$, returns exactly this input $x$. Second, if $f^{-1}$ applied to a value $y$ returns $x$, then $f(x) = y$; if $f^{-1}$ cannot compute the inverse of $y$ and returns the error symbol, then there is no $x$ such that $f$ could have returned $y$. These two assumptions are used for functions declared with `[data]`, and for serialization and deserialization functions of a type.

For a function $f$ taking $n$ parameters of types $t_i, 1 \leq i \leq n$, and returning a value of type $t$, the first assumption translates to the following code for the equation term (so, ignoring how the parameters are quantified for now):

```
match G_inv(f) (G_f(f) x_1 ... x_n) with
| Some (z_1, ..., z_n) → G_eq(t_1) x_1 z_1 ∧ ... ∧ G_eq(t_n) x_n z_n
| None → ⊥
```

The $\bot$ in the `None` case means that we want to prove that this case does not occur.

We use this code to generate code for *two* lemmas. One of them corresponds closely to the way CryptoVerif defines equations: all variables are universally quantified. Proving such lemmas in F* is more complex, and so we generate a helper lemma that is easier to prove. This lemma takes

the variables as parameters and thus expresses the assumption just for an instantiation of the variables. The two lemmas are as follows:

```
val lemma_i_inner (x_1:G_t(t_1)) ... (x_n:G_t(t_n)) : Lemma (
  (* code for the equation term *)
) [SMTPat ( G_inv(f) (G_f(f) x_1 ... x_n) )]

let lemma_i () : Lemma (∀ (x_1:G_t(t_1)) ... (x_n:G_t(t_n)).
  (* code for the equation term *)
) = ()
```

The helper lemma is the first one. The code is a `val` declaration of a lemma, so it still has to be proved in `M.Equations.fst`. It declares an SMT pattern, which lets F* automatically use the helper lemma to prove the actual lemma, the second one in the above code. This lemma takes no parameters, the universal quantification of the variables is inside. It is a `let` definition,[24] meaning it has to have a proof, directly. Here, the proof is just `():` F* succeeds in applying the previous lemma automatically thanks to the SMT pattern.

[24] It is allowed to have `let` definitions in an `fsti` interface file. It is exposed to the outside just like `val` declarations.

For the same function $f$, the second assumption translates to the following equation term:

```
match G_inv(f) y with
| Some (x_1, ..., x_n) → G_eq(t) (G_f(f) x_1 ... x_n) y
| None → ¬(∃ (x_1:G_t(t_1)) ... (x_n:G_t(t_n)). G_eq(t) (G_f(f) x_1 ... x_n) y)
```

We generate the following two lemmas, again with the first one being a helper lemma that is the one proved by the user:

```
val lemma_i_inner (y:G_t(t)) : Lemma (
  (* code for the equation term *)
) [SMTPat ( G_inv(f) y )]

let lemma_i () : Lemma (∀ (y:G_t(t)).
  (* code for the equation term *)
) = ()
```

The code for serialization and deserialization is a special case: $f$ is the serialization function $\mathbb{G}_{\mathsf{ser}}(t_1)$ for type $t_1$ and it takes only one parameter of this type, so $n = 1$. Its return type $t$ is $\mathbb{G}_{\mathsf{t}}(t) = \texttt{bytes}$. $f^{-1}$ is the deserialization function $\mathbb{G}_{\mathsf{deser}}(t_1)$ for the type $t_1$ and it takes one parameter of type `bytes`. The function's output is an option on one value of type $t_1$. The equality test function $\mathbb{G}_{\mathsf{eq}}(t)$ is always `eq_bytes`.

The reader might wonder why we do not use this style with helper lemmas for the explicitly defined equations discussed in Section 6.13.1, after all we said that this makes the proof easier. However, SMT patterns cannot treat occurrences of negation well, but the explicitly defined equations *can* contain negations. In future work, we could try to filter out any negations when creating the SMT pattern. For now, we decided to only generate the universally quantified version for the explicitly defined equations.

### 6.13.3 *Built-in Equational Theories*

Built-in equations are used with the keyword `equation builtin`, followed by the name of the equational theory, and its parameters. The signature of the lemmas that we generate is always the same, so we are not repeating it each time in the following:

```
val lemma_i : unit → Lemma ( (* code of the equation *) )
```

As with the other kinds of assumptions, we only translate those that only contain functions and constants that are translated, i. e., have an implementation annotation.

`equation builtin commut(`$f$`)`. For a binary function $f$ with two inputs of the same type $t$ and return type $t'$, this indicates that $f$ is commutative. We generate the following code for the inside of the lemma:

```
∀ (a:𝔾t(t)) (b:𝔾t(t)). 𝔾eq(t') (𝔾f(f) a b) (𝔾f(f) b a)
```

`equation builtin assoc(`$f$`)`. For a binary function $f$ with inputs and return of the same type $t$, this indicates that $f$ is associative. We generate:

```
∀ (a:𝔾t(t)) (b:𝔾t(t)) (c:𝔾t(t)).
  𝔾eq(t) (𝔾f(f) a (𝔾f(f) b c)) (𝔾f(f) (𝔾f(f) a b) c)
```

`equation builtin AC(`$f$`)`. For a binary function $f$ with inputs and return of the same type $t$, this indicates that $f$ is commutative *and* associative. We generate two separate lemmas, one for commutativity, and one for associativity, like above, respectively.

`equation builtin assocU(`$f$`, `$n$`)`. For a binary function $f$ with inputs and return of the same type $t$, and a constant $n$ also of type $t$, this indicates that $f$ is associative and that $n$ is a neutral element for $f$. We generate one lemma for associativity like above, and one for the neutral element as follows:

```
∀ (a:𝔾t(t)). (𝔾eq(t) (𝔾f(f) a 𝔾c(n)) a) ∧ (𝔾eq(t) (𝔾f(f) 𝔾c(n) a) a)
```

`equation builtin ACU(`$f$`, `$n$`)`. This is like `assocU`, adding that $f$ is also commutative. We generate three lemmas like above, for commutativity, associativity, and the neutral element.

`equation builtin ACUN(`$f$`, `$n$`)`. This is like `ACU`, adding that the equation $f(a,a) = n$ holds for all $a$ of type $t$. We generate three lemmas like above, for commutativity, associativity, and the neutral element, and a fourth one for the cancellation equation:

```
∀ (a:𝔾t(t)). 𝔾eq(t) (𝔾f(f) a a) 𝔾c(n)
```

`equation builtin group(`$f$`, `$\mathrm{inv}$`, `$n$`)`. For a binary function $f$ with inputs and return of the same type $t$, and a constant $n$ also of type $t$, and a unary function $\mathrm{inv}$ with input and return type $t$, this indicates that $f$ forms a group in type $t$, where $n$ is the neutral element and $\mathrm{inv}$ computes the inverse of an element. We generate an associativity lemma, one for the neutral element, and the following for the inverse:

```
∀ (a:𝔾t(t)).
  (𝔾eq(t) (𝔾f(f) a (𝔾f(inv) a)) 𝔾c(n)) ∧
  (𝔾eq(t) (𝔾f(f) (𝔾f(inv) a) a) 𝔾c(n))
```

`equation builtin commut_group(`$f$`, `$\mathrm{inv}$`, `$n$`)`. This is like `group`, adding that the group is commutative. We generate the same lemmas as for `group`, and a commutativity lemma for $f$ like above.

In case we know that $f$ is commutative, we could simplify the lemmas for the neutral element and the inverse, including only one of the conditions, respectively. For simplicity of our code, we chose not to do that at the moment.

For equality test functions, CryptoVerif uses the built-in equation system to indicate that they are commutative. Thus, as a side effect of our translation of built-in equations, we generate commutativity lemmas for the equality test functions of extracted types. We could prove more properties about equality test functions, and we plan to do so in future work, see Section 6.17.

For the fixed types, cv2fstar is not generating lemmas about equality test functions. However, as equality check for fixed types is directly using HACL⋆'s equality test `lbytes_eq`, commutativity and other properties are already proven within HACL⋆: `lbytes_eq` is proven to be equivalent to F⋆'s `(==)` operator.

## 6.14 Differences to the OCaml Extraction

The major difference between cv2fstar and OCaml extraction developed previously [CB13; CB15] is that cv2fstar connects CryptoVerif to a proof-oriented programming language with a richer type system, F⋆, that allows to not only execute the CryptoVerif models but also to prove more properties about them.

[CB13] Cadé and Blanchet, "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH"

[CB15] Cadé and Blanchet, "Proved Generation of Implementations from Computationally Secure Protocol Specifications"

Additionally, cv2fstar translates more parts of a CryptoVerif model than cv2ocaml does: we translate events, which are ignored in cv2ocaml; we translate equations and other assumptions as lemmas, which are not translated by cv2ocaml.

Then, there are differences in how a CryptoVerif model is translated. In cv2ocaml, a series of oracles is expressed by closures: an oracle with following oracles returns a closure representing the next possible oracle calls. In cv2fstar, we translate each oracle as a separate function and enforce the calling order by passing around a state object holding session state, and by using session IDs. We use this style rather than closures because it is closer to the style of efficient implementations in Low⋆, the subset of F⋆ that can be extracted to C code using the tool KaRaMeL [Pro+17]. This will make it easier to prove functional equivalence of the F⋆ specifications generated by cv2fstar and a Low⋆ implementation, in future work.

[Pro+17] Protzenko et al., "Verified Low-Level Programming Embedded in F*"

The state also holds table data in memory, that is always written to and read from files in cv2ocaml. For sharing variables between modules, cv2ocaml provides a mechanism by writing to and reading from files. We do not need this kind of sharing in cv2fstar, because we require all modules to be at the top of the main process.

cv2fstar makes the handling of entropy for random sampling explicit to produce pure functional code. cv2ocaml does not make this explicit and is producing code with side effects. However, as discussed in Section 6.6.1, the OCaml backend used by cv2fstar is not ideal concerning random sampling, neither. In cv2ocaml, CryptoVerif `letfun`s used to be inlined, and we translate them as separate functions in cv2fstar. This has been backported to cv2ocaml in the meantime.

While cv2ocaml allows top-level oracles to be restricted to a single call, cv2fstar requires that all top-level oracles are under replication. Furthermore, cv2fstar requires that all top-level oracles are extracted as modules, which is not the case for cv2ocaml. The motivation and some consequences of this

are described in Section 6.4.2.

In the implementation annotations of a CryptoVerif model, cv2ocaml allows the indication of a predicate function for a type. It serves to determine if a value of an OCaml type corresponds to the type in CryptoVerif. Let us take the example of all bitstrings of length $l$ bits that start with a zero bit. In OCaml, neither the length requirement nor the restriction on the starting bit can be expressed directly with the type definition. We would use `string` as base type, and a predicate that checks the additional requirements. We do not need this mechanism in cv2fstar because F*'s rich type system allows us to express such predicates directly in the definition of the type, as refinements.

Finally, as a small last difference, cv2fstar introduces the new `use_entropy` annotation for `letfun`s that shall be translated by a predefined function.

## 6.15 Case Study: The Needham-Schroeder-Lowe Protocol

As stated in Section 2.5, the Needham-Schroeder and Needham-Schroeder-Lowe protocols have been the "Hello World" of security protocol verification methodologies since many years, and so we chose to do our first case study of cv2fstar on them, too. As cv2fstar's goal is not to produce a security proof or to find flaws in a protocol specification, but to generate code for a protocol that has been proven secure, we are using the Needham-Schroeder-Lowe (NSL) protocol as example. As a reminder, this protocol uses a public-key encryption scheme and has the goal of mutually authenticating two participants. To keep our first usage of cv2fstar simple, we use the simplified version of NSL that we introduced in Section 2.5: we remove the trusted key server and instead implement it as tables in CryptoVerif. In the following subsections, we resurface some elements of the CryptoVerif model discussed in Section 2.5 that are relevant to the cv2fstar case study, and that we are slightly adapting. The reader shall feel free to go back to Section 2.5 for a more detailed description of the CryptoVerif model. For reference, we reproduce the protocol diagram in Figure 6.2.

### 6.15.1 *CryptoVerif Model for NSL*

The CryptoVerif model we presented in Section 2.5 already has the top-level oracle structure that is required by cv2fstar: the four top-level oracles `setup`, `key_register`, `initiator`, and `responder` are in parallel composition and under replication. For cv2fstar, we add a module annotation for the extraction, respectively (see Lines 2 and 6):

```
1 let setup() =
2  Setup {
3    foreach i ≤ Qsetup do
4     setup(addr: address) :=
5       ...
6  }.
```

Following the module names in the annotations, cv2fstar will generate the files `NSL.Setup.fst(i)`, `NSL.Key_Register.fst(i)`, `NSL.Initiator.fst(i)`, and `NSL.Responder.fst(i)`.

Also, we use a public-key encryption scheme where encryption can fail, because we want to use the HPKE standard to implement it.[25]

[25] HPKE instantiated with the Diffie-Hellman-based DHKEM fails encryption if the Diffie-Hellman shared secret results in the all-zero bitstring.

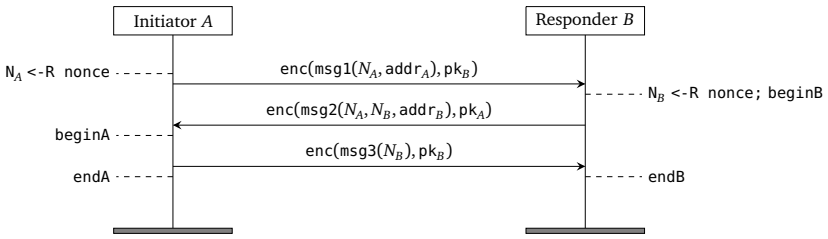



FIGURE 6.2: The simplified version of the Needham-Schroeder-Lowe protocol that we use. We assume that $A$ and $B$ possess a secret key and know the other participant's public key $\texttt{pk}_{B,A}$. We assume that they have an address $\texttt{addr}_{A,B}$ and know the one of the other participant. The events `beginA`, `beginB`, `endA`, `endB` are used to express the authentication properties of the protocol.

### 6.15.2 *Types*

In the following, we list the main types used in the CryptoVerif model and explain how we implement them in F⋆ after extraction with cv2fstar.

We define `nonce` as a fixed, large type. For key-pair generation, we use a `keyseed` that is also defined as fixed and large type. We annotate `nonce` with size 64 bits and `keyseed` with size 256 bits, and so they are translated to `lbytes 8` and `lbytes 32`. The 64 bits for `nonce` are an arbitrary choice, the 256 bits for `keyseed` correspond to the length of HPKE private keys: we want to use the HPKE standard to implement public-key encryption.

For the other types, cv2fstar generates a type declaration in `NSL.Types.fsti`. We manually write `NSL.Types.fst` to implement them, mostly by reusing the types that are defined in the cv2fstar framework in the `CVTypes` module. Concerning the implementation annotations in the CryptoVerif model, we use a convention for all types: the same name is used in F⋆, equality test functions have the name `eq_` followed by the type's name, serialization and deserialization functions are `serialize_` and `deserialize_` followed by the type's name, and random sampling functions are `gen_` followed by the type's name. While cv2fstar enforces the translation of `[fixed]` types to fixed-length types, there is no such automatism for `[bounded]` types. Therefore, in the following, we lay out how `[bounded]` types also have a bound in the F⋆ implementation.

For public-key encryption, we use the implementation of HPKE that we developed within the HACL⋆ library, see Section 5.2.5. Private keys `skey` and public keys `pkey` are declared `[bounded]` and implemented by HPKE's appropriate types; the HPKE ciphersuite we use is DHKEM with Curve25519 and SHA256, and the remainder of HPKE with the ChaCha20Poly1305 AEAD and SHA256. Curve25519 private and public keys have a fixed-size byte representation, so this is sound with respect to the `[bounded]` declaration in the CryptoVerif model, because `[fixed]` implies `[bounded]`. The type `keypair` groups together a secret and public key, we implement it by a tuple `pkey * skey`. The type `ciphertext` is `[bounded]` and implemented by a tuple of an HPKE public key (the KEM ciphertext), and a ChaCha20Poly1305 ciphertext, which is a byte sequence with an upper limit.

HPKE encryption can fail, so as described in Section 2.5.3, we instantiate a CryptoVerif public-key encryption macro such that the return type of the encryption function is an option type `ciphertext_opt`. In F⋆, we imple-

ment this option type by `option ciphertext`. cv2fstar extracts a lemma for the equation that states that the failure symbol is different from all other possible values. For F*, this lemma is trivial, and thus, it is proved without further steps. The type for `plaintext` is `[bounded]` and implemented by a ChaCha20Poly1305 plaintext, which is a byte sequence with an imposed maximum length. The type of the random coins needed for the probabilistic HPKE encryption is `encseed` and `[bounded]` in our model; in the implementation, this is an HPKE private key, and so we extract them as a fixed-length type with a size annotation of 256 bits.[26] Finally, the `[bounded]` type `address` for the participant's identities is implemented by `bytes`. We impose a maximum length derived from the maximum length of POSIX usernames and DNS hostnames.

[26] cv2fstar allows annotating bounded types as fixed-length types for the implementation.

EQUALITY TEST FUNCTIONS. The equality test functions for all these types are implemented by using `eq_bytes` internally, decomposing a tuple before, if necessary.

SERIALIZATION AND DESERIALIZATION. Serialization of a ciphertext is implemented by serializing public key and AEAD ciphertext, and concatenating both byte sequences. Deserialization splits according to the fixed size of a public key. The correctness proof reuses existing F* lemmas about splitting the result of a concatenation, and vice versa. The private and public keys and the address are already byte sequences, so serialization is the identity function. Deserialization only needs to check that the length of the input byte sequence is appropriate. F* proves their correctness without further proof indications.

RANDOM SAMPLING. The only two types that are used with random sampling are `keyseed` and `encseed`. They are both annotated as fixed-length types, and so the random sampling function is the built-in `gen_lbytes` with 32 as length parameter.

### 6.15.3 *Tables*

As a reminder, we use two tables, `trusted_keys` and `all_keys`. Each table entry contains data about a participant. The table `trusted_keys` contains addresses and key pairs of honest participants. The table `all_keys` contains addresses and the public keys of all participants, and a trust bit indicating if this participant is honest.

cv2fstar generates the following types in `NSL.Tables.fsti`. Again, we set the table implementation annotation to use the same name as in the CryptoVerif model.

```
type nsl_table_name : eqtype =
  | Table_trusted_keys
  | Table_all_keys

noeq type nsl_table_entry (tn: nsl_table_name) =
  | TableEntry_trusted_keys :
       address
    → skey
    → _: pkey{tn == Table_trusted_keys}
    → nsl_table_entry tn
```

```
| TableEntry_all_keys :
    address
  → pkey
  → _: bool{tn == Table_all_keys}
  → nsl_table_entry tn
```

In `nsl_table_entry`, we use a refinement to restrict the first constructor to the `trusted_keys` table and the second constructor to the `all_keys` table. Technically, this refinement belongs to the `pkey` and the `trust` field, respectively, so we have to give this field a name. However, as we do not use the field's name in the refinement, we can use `_`.

### 6.15.4 *Message Encoding Functions*

As a reminder, the message encoding functions `msg1`, `msg2`, and `msg3` are defined as follows:

```
fun msg1(nonce, address):plaintext [data].
fun msg2(nonce, nonce, address):plaintext [data].
fun msg3(nonce):plaintext [data].
```

We assume that the outputs of these functions never collide, by using three equations (see Section 2.5.5), that cv2fstar translates to F* lemmas. We implement the three functions in F* by a concatenation of their parameters, using a one-byte prefix `0x01`, `0x02`, `0x03` for `msg1`, `msg2`, and `msg3`, respectively. This allows us to prove the disjointness of outputs.

### 6.15.5 *Implementing Public-Key Encryption*

As a reminder, the macro we use to instantiate probabilistic public-key encryption in the CryptoVerif model declares a function `enc_r` that takes randomness explicitely as parameter, and defines a letfun `enc` that samples the randomness needed for encryption and passes it to `enc_r`:

```
fun enc_r(plaintext, pkey, encseed): ciphertext.
letfun enc(m: plaintext, pk: pkey) =
  r <-R encseed; enc_r(m, pk, r).
```

The implementation of HPKE's encryption function in HACL* expects to receive the ephemeral private key as parameter; this fits well together, because we implement the type `encseed` by a byte sequence of the right size. Thus, in F*, we implement `enc_r` as a thin wrapper around HPKE's encryption function in Base mode. The wrapper just adds the other two necessary parameters to the function call, the info bitstring and additional data, and we use an empty bitstring for both, for simplicity. We use the following implementation annotation in the CryptoVerif model:

```
implementation fun enc_r="enc".
```

The decryption function `dec_opt` takes a value of type `ciphertext_opt` as parameter, to allow it to be called on the failure value `ciphertext_bottom`. We define a letfun function `dec` as a wrapper that can be called with a value of type `ciphertext` directly. It injects the value into `ciphertext_opt` using the function `ciphertext_some` before calling `dec_opt`. We add an implementation annotation to indicate that we want to implement `dec` directly in F*, hiding the letfun's body:

```
implementation fun dec="dec".
```

We do this because the HPKE decryption function does not allow a failure symbol as parameter.

We briefly recall the CryptoVerif code for the `kp` function and the `keygen` letfun:

```
fun kp(pkey, skey): keypair [data].
letfun keygen() = k <-R keyseed; kp(pkgen(k), skgen(k)).
```

The type `keyseed` is implemented as a byte sequence of exactly the length of an HPKE private key. Thus, we implement `skgen` as the identity function, `pkgen` as exponentiation, and cv2fstar translates `keygen` as letfun doing the random sampling.

### 6.15.6 *Events and Queries*

The CryptoVerif model uses the four events `beginA`, `beginB`, `endA`, and `endB`, that are all declared to take both participants' addresses and the two nonces as parameters, as described in Section 2.5.7. cv2fstar generates the following type as part of `NSL.Events.fsti`:

```
noeq type nsl_event =
  | Event_endA : address → address → lbytes 8 → lbytes 8 → nsl_event
  | Event_beginA : address → address → lbytes 8 → lbytes 8 → nsl_event
  | Event_endB : address → address → lbytes 8 → lbytes 8 → nsl_event
  | Event_beginB : address → address → lbytes 8 → lbytes 8 → nsl_event
```

### 6.15.7 *Honest Participant Oracles*

In the following, we discuss the code generated for the initiator oracles. The code generated for the other oracles is included in Appendix D. As a reminder, the initiator process exposes three oracles that can be called only in sequence. The first oracle sends the first protocol message, the second oracle receives the second protocol message and sends the third protocol message, and the third oracle issues the event `endA` if the responder is honest.

To showcase the translation of letfuns as separate F* functions, we outsource the body of the first oracle into a letfun, deviating from the model presented in Section 2.5.8:

```
initiator_send_msg1 (addrA: address, addrX: address) :=
  let msg1succ(skA, pkX, trustX, Na, c1) =
      initiator_send_msg1_inner(addrA, addrX) in
  return (c1);
```

We declare a new option type for the result of this letfun. On the F* side, it is implemented by `option (skey * pkey * bool * nonce * ciphertext)`.

```
type msg1res_t.
fun msg1succ(skey, pkey, bool, nonce, ciphertext): msg1res_t [data].
const msg1fail: msg1res_t.
equation forall x1: skey, x2: pkey, x3: bool,
            x4: nonce, x5: ciphertext;
  msg1succ(x1, x2, x3, x4, x5) <> msg1fail.
```

The function `msg1succ` is used to construct the return value in case the encryption of the first protocol message succeeds, and the constant value `msg1fail` is used if the retrieval of key material or the encryption fails.

The body of the letfun is practically the same as the oracle body of `initiator_send_msg_1` in Section 2.5.8. However, in letfuns it is required to make the failing branch of get requests and let binding pattern matchings explicit:

```
letfun initiator_send_msg1_inner(addrA: address, addrX: address) =
 (* the gets fail if addrA or addrX have not been
    setup by the adversary. *)
 get[unique] trusted_keys(=addrA, skA, pkA) in (
  get[unique] all_keys(=addrX, pkX, trustX) in (
   (* Prepare Message 1 *)
   Na <-R nonce;
   let cc1 = enc(msg1(Na, addrA), pkX) in
   let ciphertext_some(c1: ciphertext) = cc1 in (
    msg1succ(skA, pkX, trustX, Na, c1)
   ) else msg1fail
  ) else msg1fail
 ) else msg1fail.
```

cv2fstar generates the following function declarations in `NSL.Initiator.fsti`. We see that each takes at least `nsl_state` as input and returns at least `nsl_state`. Only the first oracle returns a new session ID, and only the two last oracles take a session ID as input.

```
val oracle_initiator_send_msg1: nsl_state → address → address
 → nsl_state * option (nat * ciphertext)

val oracle_initiator_send_msg3: nsl_state → nat → ciphertext
 → nsl_state * option (ciphertext)

val oracle_initiator_finish: nsl_state → nat
 → nsl_state * option (unit)
```

The following session entry type is generated in `NSL.Sessions.fsti`. There is one entry for the responder side, used to share variables from the first oracle with the second oracle, and two entries for the initiator side. The integer suffixes of variable names starting with `var_` come from CryptoVerif, it adds them to guarantee unique variable names.

```
noeq type nsl_session_entry =
 | R1_responder_receive_msg3 :
    var_Na4: lbytes 8 →
    var_Nb2: lbytes 8 →
    var_addrB0: address →
    var_addrY0: address →
    var_skB0: skey →
    var_trustY0: bool
   → nsl_session_entry
 | R1_initiator_send_msg3 :
    var_Na3: lbytes 8 →
    var_addrA1: address →
    var_addrX1: address →
    var_pkX4: pkey →
    var_skA2: skey →
    var_trustX2: bool
   → nsl_session_entry
 | R1_initiator_finish :
    var_Na3: lbytes 8 →
    var_Nb1: lbytes 8 →
    var_addrA1: address →
    var_addrX1: address →
```

```
23      var_trustX_2: bool
24    → nsl_session_entry
```

The following code is generated for the first initiator oracle in `NSL.Initiator.fst`:

```
1  let oracle_initiator_send_msg_1 state input_43 input_42 =
2   let state, sid = state_reserve_session_id state in
3   let var_addrA_1 = input_43 in
4   let var_addrX_1 = input_42 in
5   let state, v44 = NSL.Letfun.fun_initiator_send_msg_1_inner
6                      state var_addrA_1 var_addrX_1 in
7   let bvar_45 = v44 in
8   match inv_msg1succ bvar_45 with
9   | None → state, None
10  | Some (var_skA_2, var_pkX_4, var_trustX_2, var_Na_3, var_c1_3) →
11   let sel =
12    [R1_initiator_send_msg_3 var_Na_3 var_addrA_1 var_addrX_1
13                             var_pkX_4 var_skA_2 var_trustX_2]
14   in
15   let state = state_add_to_session state sid sel in
16   (state, Some (sid, var_c1_3))
```

We can see how a session ID is reserved (Line 2), how the letfun is called (Line 5), how the letfun's result is pattern matched (Lines 7 to 10), how the session entry is created and added (Lines 11 to 15), and that the session ID is returned alongside the ciphertext (last line). This oracle does not use complex pattern matching for its parameters, and so all that is necessary is a simple assignment (Lines 3 and 4). The integer suffixes of variable names not starting with `var_` come from cv2fstar, also for the reason of creating unique variable names.

We show the beginning of the generated implementation of the second initiator oracle, to see how session entry retrieval looks like:

```
1  let oracle_initiator_send_msg_3 state sid input_46 =
2   match get_and_remove_session_entry
3           state Oracle_initiator_send_msg_3 sid with
4   | state, None → state, None
5   | state, Some (se: nsl_session_entry) →
6    match se with
7    | R1_initiator_send_msg_3 var_Na_3 var_addrA_1 var_addrX_1
8                              var_pkX_4 var_skA_2 var_trustX_2 →
```

The oracle needs a session entry matching the oracle name `Oracle_initiator_send_msg_3`; the entry is removed after retrieval because the oracle is not under replication (Line 2). The first initiator oracle has only one return, and so there is only one possible session entry (Line 7). The remaining code is included in Appendix D.

### 6.15.8 *Executing the Protocol*

As a way to evaluate the code generated by cv2fstar, we write a small test application. It runs one full protocol execution by calling the oracle functions one after another, feeding back the protocol messages without passing via a network, and printing parts of the state before and after each function call. This means we execute both the initiator and the responder within one process on one machine. While this does not constitute a real-world use case, it allows us to test if the protocol code generated by cv2fstar works.

We write an F* file `Application.fst` and extract it to OCaml using F*'s standard toolchain and a helper Makefile generated by cv2fstar.

We setup two honest participants with addresses `A@localhost` and `B@localhost` using the setup oracle. Afterwards, we use the printing helper generated by cv2fstar to print the table entries of both tables. Then, we call the first initiator oracle function with the two addresses as parameters, and go on with the other oracle functions, switching between responder and initiator. After each oracle function call, we use the generated printing helpers to print the session entries, and the protocol message. At the end, we print the list of events issued during protocol execution.

The experiment is successful: a full protocol run finishes such that both participants end up with the same two nonces, and we see that tables, sessions, and event list are populated as expected. The main code of `Application.fst` and its full printout is included in Appendices D.6 and D.7.

The sequence of oracle function calls is contained in the function `testrun` in `Application.fst`. It has effect `ML`, which means it has side effects; this is because we print information about the protocol run to the terminal. The function does not have any other side effects – after removing the printing, the `ML` effect could be removed. The function `testrun` takes an initial entropy value as parameter, which it uses to initialize the cv2fstar framework. This means the function is generic and we could prove properties about it for all entropy values. The initialization to the special `entropy0` value happens in the function `main` in `Application.fst`, which passes it as parameter to `testrun`.

We can provide an anecdotic evidence that executing code generated with cv2fstar helps finding bugs in the CryptoVerif model: While modifying the CryptoVerif model over and over again for testing different styles, we ended up using the wrong public key to encrypt one of the protocol messages. This rendered the protocol effectively incorrect; the recipient of the message would not be able to decrypt and would abort the session. CryptoVerif did still prove the correspondance queries because the properties hold also if there is no `endA` nor `endB` event. When executing the code generated by cv2fstar, it became clear that a decryption always fails, and we eventually found the mistake in `nsl.ocv`. So, the code served as test for correctness of the protocol.

## 6.16 CONCLUSION

Given the case study presented in the previous section, we can say that cv2fstar succeeds in extracting a protocol model with all oracles exposed to the adversary. Furthermore, we extract all relevant non-cryptographic assumptions to lemmas as proof obligations on the F* side, and prove them manually for our implementation. For NSL, these are the assumptions that the message decoding functions are correct inverses of the message encoding functions, the assumptions that the message encoding functions produce disjoint outputs, and the assumptions that (de)serialization functions of protocol-specific types are correct.

To conclude, cv2fstar allows to execute CryptoVerif models, and makes it possible to use a verified cryptographic library, through its framework made ready to use HACL⋆. Furthermore, because F⋆ is a proof-oriented programming language, we are not only able to fill in the implementation details that a CryptoVerif model is assuming, but can also prove that the implementation fulfills the model's assumptions. cv2fstar generates F⋆ lemmas as proof obligations for this, translating non-cryptographic assumptions in a meaningful way. The correctness of (de)serialization functions included in the cv2fstar framework for CryptoVerif's built-in types is proven once and for all, and we can reuse these lemmas to prove correctness of protocol-specific functions.

The only missing piece to have a translation of all parts of a CryptoVerif model, is the translation of the theorems proved by CryptoVerif into a form usable by F⋆, e. g., assumed lemmas.

## 6.17 Future Work

Avenues for future work on cv2fstar are manifold. From a scientific view point, the most interesting steps are those further reinforcing the formal link between CryptoVerif and F⋆. Most importantly, that would be to express the theorems that CryptoVerif proves in a way that F⋆ understands and that can be used to built upon for further proofs. For correspondance properties, the straightforward first step is to implement the definitions of non-injective and injective correspondences used by CryptoVerif [Bla07] as predicates in F⋆, and generating appropriate assumed lemmas based on it from cv2fstar. Secrecy and equivalence properties from CryptoVerif are relational properties, which are less straightforward to express and work with in F⋆. One idea is to extract both the real and ideal version of the game, and then use the real version for linking with an implementation, and the ideal version for further proofs. Another, more advanced idea would be to use a framework for relational verification like the one introduced by [Gri+19] and generate assumed lemmas that can then be used for further proofs.

[Bla07] Blanchet, "Computationally Sound Mechanized Proofs of Correspondence Assertions"

[Gri+19] Grimm et al., *A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations*

A minor next step is to generate stronger lemmas as proof obligations for equality test functions: instead of just commutativity as side effect of our translation of built-in equational theories, we could explicitly generate lemmas that establish equality test functions as being equivalent to F⋆'s `(==)` operator.

As a reinforcement of the trust in cv2fstar itself, we believe that the soundness proof of cv2ocaml [CB15] could be adapted to cv2fstar. The most important difference of cv2fstar's code generation relevant to the proof is the usage of sessions instead of closures for sequences of oracles.

[CB15] Cadé and Blanchet, "Proved Generation of Implementations from Computationally Secure Protocol Specifications"

An exciting next step would be to apply cv2fstar to a practically relevant real-world protocol. The natural next example given the work presented in this thesis would be WireGuard. Moreover, we could try to link the specification generated by cv2fstar to the efficient low-level implementation of Noise* [Ho+22], presenting an end-to-end case study from cryptographic proofs down to verified C code.

[Ho+22] Ho et al., "Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations"

There are several ideas we have to strengthen the cv2fstar framework,

using the F* type system to express more guarantees. We would like to write predicates expressing that an oracle function only adds session entries that are valid for the following oracles, and other invariants concerning the state. cv2fstar could then generate lemmas or post-conditions for the extracted oracle functions using these predicates. We could build stronger lemmas based on this, to prove that given a valid session ID, if an oracle function succeeds, all new session entries added by it are valid. This will permit to prove, using the type system, that a sequential execution of all protocol functions (oracle calls) is in principle possible.[27]

[27] This would still not catch a protocol correctness bug like the one mentioned in the previous section.

We want to mention two possible functionality improvements that we envision for the cv2fstar framework. First, we want to add support for writing tables to and reading them from a file. Second, as already mentioned in Section 6.6, we could use a cryptographically secure pseudo-random number generator like HMAC-DRBG to improve the efficiency of random sampling.

Finally, there are various improvements concerning usability that we consider. First, we would like to generate speaking lemma names instead of just numbering them. The NSL case study has shown that this is a pain point for work-in-progress models: Changing the order of type or function declarations in the CryptoVerif model influences the order in which lemmas are generated, and thus, it changes their name. This is cumbersome to fix in `NSL.Equations.fst`. Second, we might change the handling of implementation annotations. In the NSL case study, we followed a convention for all of them, which brought up the idea whether the annotations could be optional for most cases and use the convention as a default in absence of an annotation. Third, as an improvement concerning the integration of the HACL* library, we could prepare an F* module for each macro included in CryptoVerif's default library.

# Part IV

# EPILOGUE

# 7

# *Conclusion*

In this thesis, we have shown that writing cryptographic proofs using the CryptoVerif proof assistant is practical for the analysis of real-world protocols. We demonstrated this on the examples of the WireGuard VPN protocol and the HPKE standard. In our analyses, we used more detailed models of elliptic curves than any handwritten protocol analysis did before. We hope that our work on these models gets picked up and will be used by others, particularly for protocols that use X25519 or X448 for Diffie-Hellman key agreement. The advent of the ristretto255 and decaf448 groups, currently being standardized by the CFRG [Val+22], might render this less relevant for protocols developed in the future: they provide a prime-order-group abstraction layer around Curve25519 and Curve448, respectively, with only "minor overhead", and "still allowing the use of high-performance curve implementations internally". However, given the widespread support of X25519 in cryptographic libraries, we expect this algorithm to still be relevant in practice for a while.

[Val+22] Valence et al., *The ristretto255 and decaf448 Groups*

In the WireGuard analysis, we carefully modeled the protocol very close to its actual implementation: we cared in detail about the instantiation of hash and key derivation functions, and considered that X25519 Diffie-Hellman agreement is not using a prime-order group.

In the analysis of HPKE's authenticated mode, we go further and provide concrete security bounds. While the WireGuard analysis took place *after* the protocol was already deployed, the analysis of HPKE was conducted *during* development of the standard. We could provide valuable feedback to the authors, suggested significant design changes that have been incorporated, and in the end became an official co-author of the standard. The two most important design changes were to make the Diffie-Hellman-based KEM IND-CCA-secure on its own, and to introduce thorough oracle separation through labels in hash function calls.

With cv2fstar, we introduced a compiler from CryptoVerif models to executable F⋆ specifications. This allows to guarantee cryptographic properties on executable code. Additionally, because F⋆ is a proof-oriented programming language, or, a proof assistant, we can prove properties about the specific implementation of functions that are only assumed in the CryptoVerif model. This compiler represents another step in connecting analysis tools.

## 7.1 PLACEMENT OF MECHANIZED CRYPTOGRAPHIC PROOFS WITHIN THE CRYPTOGRAPHIC COMMUNITY

As mentioned in the introduction, proofs have not always been part of publications within the cryptography community. It has been an important advancement of the community's scientific standards that this is now the case. For the fact that cryptography actually *can* have mathematical proofs, it "is held up as a role model for Science in Security" [Hv17]. Nevertheless, multiple community members have called out a "crisis of rigor", with proofs being "essentially unverifiable" [BR06]. Thus, maybe it is time to go the next step and bring computer-aided cryptographic proofs to a broader audience. Halevi's motivational vision published in the year 2005 clearly has not been realized [Hal05]:

> Also, I believe that if a tool like that is built well, it will be adopted and used by many. Wouldn't you like to be cited by half of the papers appearing in CRYPTO 2010? Here is your chance...

[Hv17] Herley and van Oorschot, "SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit"

[BR06] Bellare and Rogaway, "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs"

[Hal05] Halevi, *A plausible approach to computer-aided cryptographic proofs*

It is debatable whether this was even a reachable goal: the current landscape of cryptographic proof assistants suggests that it is not expectable to have *one* tool that can handle a variety of proof methodologies or security notions, be it game-based, simulation-based, using the UC framework, or the state-separating proofs technique. However, it is clear that even after 17 years we have not even come close to Halevi's vision. As anecdotic evidence, our paper on the HPKE analysis at Eurocrypt 2021 was the only one using a proof assistant. It seems Rogaway's statement from 2015 is still true: "formalistic approaches are gone [from cryptography's tier-1 venues] (unless they claim to bridge to 'real' crypto)" [Rog15]. We believe this is because building a cryptographic proof assistant that is expressive enough to handle practically relevant proofs *and* is user-friendly is truly hard and requires an extensive engineering effort. Furthermore, as Halevi already said in 2005, developing a user-friendly proof assistant "does not have a very appealing 'business case'" [Hal05]. With probably only cryptographers as a relatively small user base for any tool in this space, we have to navigate incentives and funding sources for an effort in making proof assistants more user-friendly. If we want more users, we clearly have to invest in such an effort, because currently, only people in the immediate surroundings of tool developers are able to use cryptographic proof assistants efficiently and effectively. Another option would be to rely on the tool developers and their surroundings as "proof engineers" that can be brought into or contracted for a project – however, it is doubtful whether this approach would scale sufficiently well within the academic cryptography community. [1] There is a clear demand for computer-assisted proofs, as shown by cryptographers reaching out expressing their interest in using a tool, and standardization bodies being interested in seeing high-assurance proofs of cryptographic systems currently under standardization.

[Rog15] Rogaway, *The Moral Character of Cryptographic Work*

[1] The term "proof engineer(ing)" was coined by the sel4 verification project: https://trustworthy.systems/projects/seL4-verification/.

DO TOOLS ACTUALLY HELP DETECT ERRORS? In his two articles "Another Look at Automated Theorem-Proving" [Kob07; Kob12], Neal Koblitz asks if proof assistants actually help catch errors of the type that have been

[Kob07] Koblitz, "Another look at automated theorem-proving"

[Kob12] Koblitz, "Another look at automated theorem-proving II"

embarrassing for the cryptographic community in the past. We want to mention two cases of proof mistakes in connection to the work presented in this thesis.

First, for the case of WireGuard, we recall that a handwritten proof was published by Dowling and Paterson [DP18] before our mechanized proof. In a work on a post-quantum-secure version of WireGuard, the authors Hülsing, Ning, Schwabe, Weber, and Zimmermann [Hül+21] build upon the proof by Dowling and Paterson. Weber found a mistake in the original proof, which consists in the accidental swapping of two parameters of a KDF, and led to the realization that a dual PRF assumption is needed to regain a proof. [2] Such a mistake would probably have been found using a proof assistant like CryptoVerif or EasyCrypt: either by the type system in case the parameters of the KDF have different types, or when trying to apply the cryptographic hardness assumption used for the KDF; using a parameter as KDF key would be impossible in CryptoVerif if it has not been shown to be indistinguishable from a randomly sampled value, amongst other conditions. We could not find this error because we did not formalize Dowling and Paterson's proof which uses the PRF-ODH assumption in the standard model. We wrote our proof in the random oracle model because at that time, CryptoVerif was not yet equipped to handle PRF-ODH assumptions. (Also, it seems implausible to instantiate PRF-ODH without using a random oracle [Bre+17].) We show that HKDF-BLAKE2 is a random oracle (assuming that BLAKE2 is a random oracle), which implies the dual PRF assumption.

[DP18] Dowling and Paterson, "A Cryptographic Analysis of the WireGuard Protocol"

[Hül+21] Hülsing et al., "Post-quantum WireGuard"

[2] See Section IV-A in [Hül+21].

[Bre+17] Brendel et al., "PRF-ODH: Relations, Instantiations, and Impossibility Results"

Second, we recall that our analysis of HPKE's Auth mode that was published at Eurocrypt 2021 contains several handwritten proofs in the appendix that establish our theorems for 1-user and 2-user security notions. We can report that an ongoing formalization effort of these proofs by Pierre Boutry and Christian Doczkal using EasyCrypt surfaced a few subtle errors concerning collisions of public keys and ciphertexts, and the counting of adversary queries to oracles exposed by the game. These errors have been corrected in the long version of the paper [Alw+20] and in this thesis.

[Alw+20] Alwen et al., *Analysing the HPKE Standard*

While it is debatable if these examples constitute embarrassing mistakes, we believe, in response to Koblitz, that formal methods and proof assistants *do* deliver on their promises to detect errors and to produce proofs with a high assurance.

In their survey about the Science of Security, Herley and van Oorschot cite DeMillo, Lipton, and Perlis, who express the opinion that (1) "it is a social process that determines whether mathematicians feel confident about a theorem", that (2) "no comparable social process can take place among program verifiers", and that (3) "the point is that mathematicians' errors are corrected, not by formal symbolic logic, but by other mathematicians".

Responding to the first and second quote, we believe that the building up of trust into a proof assistant can be a *separate* social process from the one determining confidence about a theorem. And we believe that building up trust into a proof assistant is actually a social process that *can* take place, just like proof methodologies and styles in cryptography have shifted over the years, e. g., the cultural shift to code-based proofs. This social process has not yet converged for CryptoVerif, as we can report on the example of the HPKE analysis: when we started working with our co-authors, it was

hard for them to even understand the *statements* of the theorems proven in CryptoVerif, because its input language was unknown to them and not close enough to the style known from cryptography papers to be understandable without further explanation.

For a specific proof conducted using a proof assistant, there is still place for a social process to determine whether the community feels confident about the formulation of the theorem in the proof assistant – the benefit of using the proof assistant is that the social process only needs to be about the input to the proof assistant, and not about checking the correctness of the entire proof; of course given that the proof assistant has generally already gained trust. On the specific example of HPKE, we present our theorems and security notions in a way typical for cryptography venues – part of the social process, or review, should then be the confirmation that this representation is equivalent to the implementation in CryptoVerif.

Responding to the third quote, we just want to say that the proof assistants do not act on their own, they are tools *used* by mathematicians or cryptographers that help *them* find errors. At least currently, this seems clear to us, because cryptographic proofs assistants are still expert tools, in the sense that the user has to understand the proof methodology to use the tool in a meaningful way. This might be different from symbolic analysis tools that exhibit a high degree of automation, more or less hiding the underlying formalism.

PROOF ASSISTANTS AS A FORCING FUNCTION? In the conclusion of his first essay on automated theorem proving [Kob07], Koblitz asks what the benefit of a proof assistant is, compared to a handwritten proof, "written out clearly and carefully with proper attention to lucid expository style". This looks like another social effect to us: there rarely seem to be handwritten proofs actually written out in great detail. A proof assistant can be a *forcing function* here, because by design, they require a great level of detail. The notion of a forcing function was used by [Hv17] as a suggestion to resolve the problem of "undocumented and implicit assumptions [that] are common in security research": "one possibility is to find a *forcing function* to make assumptions explicit". For cryptography, such a forcing function could be to require computer-verified proofs or at least establish a culture of using proof assistants more. [3] Then, starting from assumptions underlying the proof assistant, the formal language used by the tool forces users to document everything.

[Kob07] Koblitz, "Another look at automated theorem-proving"

[Hv17] Herley and van Oorschot, "SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit"

[3] Such a culture is for example present in the POPL community, where it is much more common to formalize the semantics of a newly proposed (research) programming language using a proof assistant.

ARTIFACT EVALUATION. With mechanized proofs, reproducibility becomes interesting: as proof assistants are evolving, it is not assured that a once-published proof can still be checked by more recent versions of the proof assistant, or, an even more basic requirement, if readers of a paper manage to install the tool and re-check the proof. For now, the community relies on the original authors to publish the original files in a suitable way. When mechanized proofs are getting more popular, we encourage publication venues and the community to adopt the artifact evaluation approach that has been used by the POPL community since 2015 [4] and by the CHES community since 2021 [5]. The main idea behind artifact evaluation is to

[4] https://popl22.sigplan.org/track/POPL-2022-artifact-evaluation

[5] https://ches.iacr.org/2022/artifacts.php

ensure reproducibility and reusability of the code that underlines the results presented in a paper.

## 7.2 Future Work

We plan to continue working with cryptographic proof assistants in the area of practice-oriented cryptography both by collaborating with cryptographers on new constructions, and by supporting formal and informal standardization efforts, like within the IETF and IRTF, and for example the Noise protocol framework community. Specific protocols that could be targets of mechanized proofs are the Messaging Layer Security secure group messaging protocol [6] and some of the ongoing standardization efforts within the IRTF's CFRG. [7]

Furthermore, we plan to advance the development of proof assistants on different fronts. On the one hand, improving usability seems crucial to bring mechanized proofs to a wider audience, and we would like to support efforts in this direction. This could be trying to bring some kinds of automation to EasyCrypt, or developing a user-friendly frontend for CryptoVerif or EasyCrypt, inspired by, e. g., the Visual Studio Code extension for the Lean theorem prover, to mention just two ideas. [8] On the other hand, the adaption of proof assistants such that they are able to handle security proofs considering quantum adversaries is an important research direction. There have been first steps done for EasyCrypt [Bar+21] and Squirrel [CFJ22]. Case studies for mechanized proofs could be the candidates of the NIST post-quantum cryptography competition, [9] and post-quantum-secure versions of important protocols, in the spirit of KEMTLS [SSW20] and PQ-WireGuard [Hül+21]. Post-quantum security clearly fits into Rogaway's call for anti-surveillance research, as the first users of large-scale quantum computers might well be state agencies or other global players.

We are looking forward to the first results of the ongoing work on a translation between CryptoVerif and EasyCrypt, and are interested in exploring more potential links between proof assistants. Concretely, we want to finish up some of the ideas mentioned as future work in the cv2fstar chapter, and apply the compiler to our WireGuard models written in CryptoVerif.

Finally, to conclude this thesis, we believe that computer-aided cryptographic proofs are an exciting and in-demand research and development effort, with many promising open problems.

[6] https://messaginglayersecurity.rocks/

[7] https://datatracker.ietf.org/rg/cfrg/documents/

[8] https://github.com/leanprover/vscode-lean

[Bar+21] Barbosa et al., "EasyPQC: Verifying Post-Quantum Cryptography"

[CFJ22] Cremers et al., "A Logic and an Interactive Prover for the Computational Post-Quantum Security of Protocols"

[9] https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

[SSW20] Schwabe et al., "Post-Quantum TLS Without Handshake Signatures"

[Hül+21] Hülsing et al., "Post-quantum WireGuard"

# *Bibliography*


[AF01] Martín Abadi and Cédric Fournet. "Mobile Values, New Names, and Secure Communication". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. London, United Kingdom: Association for Computing Machinery, 2001, pp. 104115. ISBN: 1581133367. DOI: 10.1145/360204.360213. URL: https://doi.org/10.1145/360204.360213 (cit. on pp. 14, 15).

[Aba+21] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq". In: *CSF 2021 Computer Security Foundations Symposium*. Ed. by Ralf Küsters and Dave Naumann. IEEE Computer Society Press, 2021, pp. 1–15. DOI: 10.1109/CSF51468.2021.00048 (cit. on p. 6).

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. "The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES". In: *CT-RSA 2001*. Ed. by David Naccache. Vol. 2020. LNCS. Springer, Heidelberg, Apr. 2001, pp. 143–158. DOI: 10.1007/3-540-45353-9_12 (cit. on p. 83).

[AGJ11] Mihhail Aizatulin, Andy Gordon, and Jan Jürjens. "Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution". In: *CCS '11 Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM Press, 2011, pp. 331–340. ISBN: 978-1-4503-0948-6. URL: https://www.microsoft.com/en-us/research/publication/extracting-verifying-cryptographic-models-c-protocol-code-symbolic-execution/ (cit. on p. 128).

[AGJ12] Mihhail Aizatulin, Andy Gordon, and Jan Jürjens. *Computational Verification of C Protocol Implementations by Symbolic Execution*. Tech. rep. MSR-TR-2012-80. CCS '12 Proceedings of the 2012 ACM conference on Computer and communications security. 2012, pp. 712–723. URL: https://www.microsoft.com/en-us/research/publication/computational-verification-of-c-protocol-implementations-by-symbolic-execution/ (cit. on p. 128).

[Alm+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-Assurance and High-Speed Cryptography". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 18071823. ISBN: 9781450349468. DOI: 10.1145/3133956.3134078. URL: https://doi.org/10.1145/3133956.3134078 (cit. on p. 128).

[Alm+19] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. "Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 16071622. ISBN: 9781450367479. DOI: 10.1145/3319535.3363211. URL: https://doi.org/10.1145/3319535.3363211 (cit. on pp. 5, 128).

[Alw+] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. *Analysing the HPKE Standard Supplementary Material*. DOI: 10.5281/zenodo.4297811 (cit. on pp. 100, 102, 103, 106, 224, 227).

[Alw+20] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. *Analysing the HPKE Standard*. Cryptology ePrint Archive, Report 2020/1499. https://eprint.iacr.org/2020/1499. 2020 (cit. on pp. 78, 183).

[Alw+21] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. "Analysing the HPKE Standard". In: *EUROCRYPT 2021, Part I*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. LNCS. Springer, Heidelberg, Oct. 2021, pp. 87–116. DOI: 10.1007/978-3-030-77870-5_4 (cit. on pp. 8, 78).

[Aum+13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: Simpler, Smaller, Fast as MD5". In: *Applied Cryptography and Network Security*. Vol. 7954. LNCS. Springer, 2013, pp. 119–135 (cit. on p. 58).

[BHU09] Michael Backes, Dennis Hofheinz, and Dominique Unruh. "CoSP: A General Framework for Computational Soundness Proofs". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 6678. ISBN: 9781605588940. DOI: 10.1145/1653662.1653672. URL: https://doi.org/10.1145/1653662.1653672 (cit. on p. 128).

[BHM14] Michael Backes, Catalin Hritcu, and Matteo Maffei. "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations". In: *J. Comput. Secur.* 22.2 (2014), pp. 301–353. DOI: 10.3233/JCS-130493. URL: https://doi.org/10.3233/JCS-130493 (cit. on p. 129).

[Bae+21] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. "An Interactive Prover for Protocol Verification in the Computational Model". In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 537–554. DOI: 10.1109/SP40001.2021.00078 (cit. on p. 6).

[BC14] Gergei Bana and Hubert Comon-Lundh. "A Computationally Complete Symbolic Attacker for Equivalence Properties". In: *ACM CCS 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM Press, Nov. 2014, pp. 609–620. DOI: 10.1145/2660267.2660276 (cit. on p. 6).

[Bar+21] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. "EasyPQC: Verifying Post-Quantum Cryptography". In: *ACM CCS 2021*. Ed. by Giovanni Vigna and Elaine Shi. ACM Press, Nov. 2021, pp. 2564–2586. DOI: 10.1145/3460120.3484567 (cit. on p. 185).

[BAC19] Richard L. Barnes, Joël Alwen, and Sandro Corretti. *Homomorphic Multiplication for X25519 and X448*. Internet Draft. https://tools.ietf.org/html/draft-barnes-cfrg-mult-for-7748-00. IETF, Nov. 2019 (cit. on p. 92).

[Bar+20a] Richard L. Barnes, Benjamin Beurdouche, Jon Millican, E. Omara, K. Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft draft-ietf-mls-protocol-09. IETF Secretariat, Mar. 2020. URL: https://tools.ietf.org/html/draft-ietf-mls-protocol-09 (cit. on p. 79).

[Bar+20b] Richard L. Barnes, Karthik Bhargavan, Benjamin Lipp, and Christopher A. Wood. *Hybrid Public Key Encryption*. Internet-Draft draft-irtf-cfrg-hpke-08. IETF Secretariat, Oct. 2020. URL: https://tools.ietf.org/html/draft-irtf-cfrg-hpke-08 (cit. on p. 79).

[Bar+22] Richard L. Barnes, Karthik Bhargavan, Benjamin Lipp, and Christopher A. Wood. *Hybrid Public Key Encryption*. RFC 9180. RFC Editor, Feb. 2022, pp. 1–107. URL: https://www.rfc-editor.org/rfc/rfc9180.html (cit. on pp. 9, 78, 79, 102, 106, 119).

[Bar+14] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. "Probabilistic Relational Verification for Cryptographic Implementations". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 193205. ISBN: 9781450325448. DOI: 10.1145/2535838.2535847. URL: https://doi.org/10.1145/2535838.2535847 (cit. on p. 128).

[Bar+11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 71–90. ISBN: 978-3-642-22792-9 (cit. on pp. 5, 14, 128).

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. "Formal Certification of Code-Based Cryptographic Proofs". In: *SIGPLAN Not.* 44.1 (2009), pp. 90101. ISSN: 0362-1340. DOI: 10.1145/1594834.1480894. URL: https://doi.org/10.1145/1594834.1480894 (cit. on pp. 5, 127).

[Bel15] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision Resistance". In: *Journal of Cryptology* 28.4 (Oct. 2015), pp. 844–878. DOI: 10.1007/s00145-014-9185-x (cit. on p. 105).

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication". In: *CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. LNCS. Springer, Heidelberg, Aug. 1996, pp. 1–15. DOI: 10.1007/3-540-68697-5_1 (cit. on p. 105).

[BDG20] Mihir Bellare, Hannah Davis, and Felix Günther. "Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability". In: *EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. LNCS. Springer, Heidelberg, May 2020, pp. 3–32. DOI: 10.1007/978-3-030-45724-2_1 (cit. on p. 118).

[BN00] Mihir Bellare and Chanathip Namprempre. "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm". In: *ASIACRYPT'00*. Vol. 1976. LNCS. Springer, Dec. 2000, pp. 531–545 (cit. on pp. 43, 116).

[BR93] Mihir Bellare and Philip Rogaway. "Random Oracles Are Practical: a Paradigm for Designing Efficient Protocols". In: *ACM CCS'93*. ACM Press, 1993, pp. 62–73 (cit. on p. 43).

[BR04] Mihir Bellare and Phillip Rogaway. *Code-Based Game-Playing Proofs and the Security of Triple Encryption*. Cryptology ePrint Archive, Report 2004/331. https://eprint.iacr.org/2004/331. 2004 (cit. on pp. 5, 11, 13, 85).

[BR06] Mihir Bellare and Phillip Rogaway. “The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs”. In: *EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. LNCS. Springer, Heidelberg, 2006, pp. 409–426. DOI: 10.1007/11761679_25 (cit. on pp. 4, 182).

[BS20] Mihir Bellare and Igors Stepanovs. “Security Under Message-Derived Keys: Signcryption in iMessage”. In: *EUROCRYPT 2020, Part III*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12107. LNCS. Springer, Heidelberg, May 2020, pp. 507–537. DOI: 10.1007/978-3-030-45727-3_17 (cit. on p. 84).

[BT16] Mihir Bellare and Björn Tackmann. “The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3”. In: *CRYPTO 2016, Part I*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9814. LNCS. Springer, Heidelberg, Aug. 2016, pp. 247–276. DOI: 10.1007/978-3-662-53018-4_10 (cit. on p. 109).

[Ben+08] Jesper Bengtson, Karthik Bhargavan, Cédric Fournet, Andy Gordon, and Sergio Maffeis. *Refinement Types for Secure Implementations*. Tech. rep. MSR-TR-2008-118. 20th IEEE Computer Security Foundations Symposium (CSF) 2008. 2008, pp. 17–32. URL: https://www.microsoft.com/en-us/research/publication/refinement-types-for-secure-implementations/ (cit. on p. 129).

[Ber+15] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. “Verified Correctness and Security of OpenSSL HMAC”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 207–221. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer (cit. on pp. 6, 128).

[Ber05] Daniel J. Bernstein. “The Poly1305-AES Message-Authentication Code”. In: *FSE 2005*. Vol. 3557. LNCS. Springer, 2005, pp. 32–49 (cit. on p. 43).

[Ber06] Daniel J. Bernstein. “Curve25519: New Diffie-Hellman Speed Records”. In: *PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. LNCS. Springer, Heidelberg, Apr. 2006, pp. 207–228. DOI: 10.1007/11745853_14 (cit. on pp. 44, 88).

[Ber11] Daniel J. Bernstein. *Extending the Salsa20 nonce*. https://cr.yp.to/snuffle/xsalsa-20110204.pdf. 2011 (cit. on p. 41).

[Beu+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. “A Messy State of the Union: Taming the Composite State Machines of TLS”. In: *IEEE S&P (Oakland)*. 2015, pp. 535–552 (cit. on p. 34).

[Bha+08a] Karthik Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zalinescu. “Cryptographically verified implementations for TLS”. In: *15th ACM Conference on Computer and Communications Security (CCS'08)*. Association for Computing Machinery, Inc., 2008. URL: https://www.microsoft.com/en-us/research/publication/cryptographically-verified-implementations-for-tls/ (cit. on p. 128).

[Bha+17] Karthikeyan Bhargavan, Benjamin Beurdouche, Jean-Karim Zinzindohoué, and Jonathan Protzenko. “HACL*: A Verified Modern Cryptographic Library”. In: *ACM CCS* (2017). URL: https://www.microsoft.com/en-us/research/publication/hacl-a-verified-modern-cryptographic-library/ (cit. on pp. 119, 126).

[Bha+21] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. “DY* : A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code”. In: *EuroS&P 2021 - 6th IEEE European Symposium on Security and Privacy*. Virtual, Austria, Sept. 2021. URL: https://hal.inria.fr/hal-03178425 (cit. on p. 128).

[BBK17] 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*. IEEE Computer Society Press, May 2017, pp. 483–502. DOI: 10.1109/SP.2017.26 (cit. on pp. 5, 15, 74, 82, 126).

[Bha+14a] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. “Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS”. In: *IEEE S&P (Oakland)*. 2014, pp. 98–113 (cit. on pp. 34, 67).

[Bha+08b] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. “Verified Interoperable Implementations of Security Protocols”. In: *ACM Trans. Program. Lang. Syst.* 31.1 (2008). ISSN: 0164-0925. DOI: 10.1145/1452044.1452049. URL: https://doi.org/10.1145/1452044.1452049 (cit. on p. 129).

[Bha+13] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. “Implementing TLS with Verified Cryptographic Security”. In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 445–459. DOI: 10.1109/SP.2013.37 (cit. on p. 128).

[Bha+14b] Karthikeyan Bhargavan, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella-Béguelin, and Cédric Fournet. “Proving the TLS Handshake Secure (As It Is)”. In: *Advances in Cryptology – CRYPTO 2014*. Vol. 8617. Lecture Notes in Computer Science. Springer, 2014, pp. 235–255. URL: https://www.microsoft.com/en-us/research/publication/proving-the-tls-handshake-secure-as-it-is/ (cit. on p. 128).

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *ACM CCS'16*. 2016, pp. 456–467 (cit. on p. 34).

[Bla07] Bruno Blanchet. "Computationally Sound Mechanized Proofs of Correspondence Assertions". In: *CSF 2007 Computer Security Foundations Symposium*. Ed. by Andrei Sabelfeld. IEEE Computer Society Press, 2007, pp. 97–111. DOI: 10.1109/CSF.2007.16 (cit. on pp. 17, 18, 35, 176).

[Bla08] Bruno Blanchet. "A Computationally Sound Mechanized Prover for Security Protocols". In: *IEEE Transactions on Dependable and Secure Computing* 5.4 (2008), pp. 193–207 (cit. on pp. 35, 82, 126).

[Bla12] Bruno Blanchet. "Security Protocol Verification: Symbolic and Computational Models". In: *Principles of Security and Trust, POST'12*. Vol. 7215. LNCS. Springer, 2012, pp. 3–29 (cit. on p. 74).

[Bla16] Bruno Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif". In: *Foundations and Trends in Privacy and Security* 1.1-2 (Oct. 2016), pp. 1–135. ISSN: 2474-1558 (cit. on pp. 5, 34).

[Bla17] Bruno Blanchet. *CryptoVerif: A Computationally-Sound Security Protocol Verifier*. Nov. 20, 2017. URL: https://bblanche.gitlabpages.inria.fr/CryptoVerif/cryptoverif.pdf (cit. on pp. 14–17).

[Bre+17] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. "PRF-ODH: Relations, Instantiations, and Impossibility Results". In: *CRYPTO 2017, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. LNCS. Springer, Heidelberg, Aug. 2017, pp. 651–681. DOI: 10.1007/978-3-319-63697-9_22 (cit. on pp. 53, 107, 183).

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. *Cryptographic Security of the MLS RFC, Draft 11*. Cryptology ePrint Archive, Report 2021/137. https://eprint.iacr.org/2021/137. 2021 (cit. on p. 6).

[Brz+18] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. "State Separation for Code-Based Game-Playing Proofs". In: *ASIACRYPT 2018, Part III*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11274. LNCS. Springer, Heidelberg, Dec. 2018, pp. 222–249. DOI: 10.1007/978-3-030-03332-3_9 (cit. on p. 6).

[BAN90] Michael Burrows, Martin Abadi, and Roger Needham. "A Logic of Authentication". In: *ACM Trans. Comput. Syst.* 8.1 (1990), pp. 1836. ISSN: 0734-2071. DOI: 10.1145/77648.77649. URL: https://doi.org/10.1145/77648.77649 (cit. on p. 4).

[CB13] David Cadé and Bruno Blanchet. "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH". In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 4.1 (Mar. 2013). Special issue ARES'12, pp. 4–31 (cit. on pp. 126, 127, 133, 167).

[CB15] David Cadé and Bruno Blanchet. "Proved Generation of Implementations from Computationally Secure Protocol Specifications". In: *Journal of Computer Security* 23.3 (2015), pp. 331–402 (cit. on pp. 127, 133, 167, 176).

[Can00] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Report 2000/067. https://eprint.iacr.org/2000/067. 2000 (cit. on p. 5).

[CSV19] Ran Canetti, Alley Stoughton, and Mayank Varia. "EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security". In: *CSF 2019 Computer Security Foundations Symposium*. Ed. by Stephanie Delaune and Limin Jia. IEEE Computer Society Press, 2019, pp. 167–183. DOI: 10.1109/CSF.2019.00019 (cit. on p. 5).

[CD09] Sagar Chaki and Anupam Datta. "ASPIER: An Automated Framework for Verifying Security Protocol Implementations". In: *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*. CSF '09. USA: IEEE Computer Society, 2009, pp. 172185. ISBN: 9780769537122. DOI: 10.1109/CSF.2009.20. URL: https://doi.org/10.1109/CSF.2009.20 (cit. on p. 129).

[CT08] Liqun Chen and Qiang Tang. "Bilateral Unknown Key-Share Attacks in Key Agreement Protocols". In: *Journal of Universal Computer Science* 14.3 (Feb. 2008), pp. 416–440 (cit. on p. 67).

[CR10] Yannick Chevalier and Michaël Rusinowitch. "Compiling and securing cryptographic protocols". In: *Information Processing Letters* 110.3 (2010), pp. 116–122. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2009.11.004. URL: https://www.sciencedirect.com/science/article/pii/S0020019009003251 (cit. on p. 128).

[Cor+08] Ricardo Corin, Pierre-Malo Denielou, Cédric Fournet, Karthik Bhargavan, and James Leifer. "A Secure Compiler for Session Abstractions". In: *Journal of Computer Security (Special issue for CSF'07)* (2008). URL: https://www.microsoft.com/en-us/research/publication/secure-compiler-session-abstractions/ (cit. on p. 128).

[Cor+05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. "Merkle-Damgård Revisited: How to Construct a Hash Function". In: *CRYPTO 2005*. Vol. 3621. LNCS. Springer, 2005, pp. 430–448 (cit. on p. 54).

[CKW11] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. "A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems". In: *Journal of Automated Reasoning* 46.3 (2011), pp. 225–259. DOI: 10.1007/s10817-010-9187-9. URL: https://doi.org/10.1007/s10817-010-9187-9 (cit. on p. 6).

[CS03] Ronald Cramer and Victor Shoup. "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack". In: *SIAM Journal on Computing* 33.1 (2003), pp. 167–226 (cit. on p. 79).

[CFJ22] Cas Cremers, Caroline Fontaine, and Charlie Jacomme. "A Logic and an Interactive Prover for the Computational Post-Quantum Security of Protocols". In: *S&P 2022 - 43rd IEEE Symposium on Security and Privacy*. San Francisco / Virtual, United States, May 2022. URL: https://hal.inria.fr/hal-03620358 (cit. on p. 185).

[CJ19] Cas Cremers and Dennis Jackson. "Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman". In: *IEEE CSF'19*. To appear. Extended version at https://people.cispa.io/cas.cremers/downloads/papers/prime_order_please.pdf. June 2019 (cit. on p. 74).

[DL+17] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. "Implementing and Proving the TLS 1.3 Record Layer". In: *SP '17 38th IEEE Symposium on Security and Privacy*. 2017. URL: https://www.microsoft.com/en-us/research/publication/implementing-proving-tls-1-3-record-layer/ (cit. on p. 128).

[Den05a] Alexander W. Dent. "Hybrid Signcryption Schemes with Insider Security". In: *ACISP 05*. Ed. by Colin Boyd and Juan Manuel González Nieto. Vol. 3574. LNCS. Springer, Heidelberg, July 2005, pp. 253–266 (cit. on p. 84).

[Den05b] Alexander W. Dent. "Hybrid Signcryption Schemes with Outsider Security". In: *ISC 2005*. Ed. by Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao. Vol. 3650. LNCS. Springer, Heidelberg, Sept. 2005, pp. 203–217 (cit. on p. 84).

[DZ10] Alexander W. Dent and Yuliang Zheng, eds. *Practical Signcryption*. Information Security and Cryptography. Springer, 2010. ISBN: 978-3-540-89409-4. DOI: 10.1007/978-3-540-89411-7 (cit. on pp. 83, 84).

[Dod+12] Yevgeniy Dodis, Thomas Ristenpart, John Steinberger, and Stefano Tessaro. "To Hash or Not to Hash Again? (In)Differentiability Results for $H^2$ and HMAC". In: *CRYPTO 2012*. Vol. 7417. LNCS. Full version at https://eprint.iacr.org/2013/382. Springer, 2012, pp. 348–366 (cit. on pp. 58, 107, 204).

[DY83] Danny Dolev and Andrew C. Yao. "On the Security of Public Key Protocols". In: *IEEE Transactions on Information Theory* IT-29.12 (Mar. 1983), pp. 198–208. URL: https://doi.org/10.1109/TIT.1983.1056650 (cit. on p. 4).

[Don17] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *Network and Distributed System Security Symposium, NDSS*. We use the up-to-date whitepaper version for our analysis, which differs in how the MACs are defined: https://www.wireguard.com/papers/wireguard.pdf, Nov. 2nd, 2017, draft revision ceb3a49. 2017 (cit. on pp. 4, 34–36, 41, 67, 74).

[DM18] Jason A. Donenfeld and Kevin Milner. *Formal Verification of the WireGuard Protocol*. https://www.wireguard.com/papers/wireguard-formal-verification.pdf. 2018 (cit. on pp. 34, 54, 74, 75).

[DP18] Benjamin Dowling and Kenneth G. Paterson. "A Cryptographic Analysis of the WireGuard Protocol". In: *Applied Cryptography and Network Security, ACNS 2018*. Vol. 10892. LNCS. Springer, 2018, pp. 3–21 (cit. on pp. 7, 35, 53, 54, 74, 75, 183).

[DKO21] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. *Bringing State-Separating Proofs to EasyCrypt - A Security Proof for Cryptobox*. Cryptology ePrint Archive, Report 2021/326. https://eprint.iacr.org/2021/326. 2021 (cit. on p. 6).

[Dup+11] François Dupressoir, Andy Gordon, Jan Jürjens, and David A. Naumann. *Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols*. Tech. rep. MSR-TR-2011-50. Computer Security Foundations Symposium (CSF), 2011 IEEE 24th. 2011. URL: https://www.microsoft.com/en-us/research/publication/guiding-a-general-purpose-c-verifier-to-prove-cryptographic-protocols-2011/ (cit. on p. 129).

[FM21] Marc Fischlin and Arno Mittelbach. *An Overview of the Hybrid Argument*. Cryptology ePrint Archive, Report 2021/088. https://eprint.iacr.org/2021/088. 2021 (cit. on pp. 11–13, 15).

[FGR09] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. "A Security-Preserving Compiler for Distributed Programs". In: *ACM Conference on Computer and Communications Security (CCS'09)*. ACM, 2009, pp. 432–441. URL: https://www.microsoft.com/en-us/research/publication/security-preserving-compiler-distributed-programs/ (cit. on p. 128).

[FKS11] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. "Modular code-based cryptographic verification". In: *ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 341–350. ISBN: 978-1-4503-0948-6. URL: https://www.microsoft.com/en-us/research/publication/modular-code-based-cryptographic-verification/ (cit. on p. 128).

[Fro+19] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. "A Verified, Efficient Embedding of a Verifiable Assembly Language". In: *Principles of Programming Languages (POPL 2019)*. ACM, 2019. URL: https://www.microsoft.com/en-us/research/publication/a-verified-efficient-embedding-of-a-verifiable-assembly-language/ (cit. on p. 126).

[GM+10] Víctor Gayoso Martínez, F. Alvarez, Luis Hernandez Encinas, and Carmen Sánchez Ávila. "A comparison of the standardized versions of ECIES". In: *2010 6th International Conference on Information Assurance and Security, IAS 2010* (Aug. 2010). DOI: 10.1109/ISIAS.2010.5604194 (cit. on p. 83).

[GH04] Henri Gilbert and Helena Handschuh. "Security Analysis of SHA-256 and Sisters". In: *SAC 2003*. Ed. by Mitsuru Matsui and Robert J. Zuccherato. Vol. 3006. LNCS. Springer, Heidelberg, Aug. 2004, pp. 175–193. DOI: 10.1007/978-3-540-24654-1_13 (cit. on p. 105).

[Gir19] Guillaume Girol. "Formalizing and Verifying the Security Protocols from the Noise Framework". Available at https://doi.org/10.3929/ethz-b-000332859. MA thesis. ETH Zürich, Mar. 2019 (cit. on pp. 75, 76).

[GM84] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption". In: *Journal of Computer and System Sciences* 28 (1984), pp. 270–299 (cit. on p. 5).

[Gri+19] Niklas Grimm, Kenji Maillard, Cédric Fournet, Catalin Hritcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. *A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations*. 2019. arXiv: 1703.00055 [cs.PL] (cit. on pp. 128, 176).

[GJK21] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. "KHAPE: Asymmetric PAKE from Key-Hiding Key Exchange". In: *CRYPTO 2021, Part IV*. Ed. by Tal Malkin and Chris Peikert. Vol. 12828. LNCS. Virtual Event: Springer, Heidelberg, Aug. 2021, pp. 701–730. DOI: 10.1007/978-3-030-84259-8_24 (cit. on p. 6).

[Gut+05] Joshua D. Guttman, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. "Programming Cryptographic Protocols". In: *Trustworthy Global Computing*. Ed. by Rocco De Nicola and Davide Sangiorgi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 116–145. ISBN: 978-3-540-31483-7 (cit. on p. 128).

[Haa+18] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. "Computer-Aided Proofs for Multiparty Computation with Active Security". In: *CSF 2018 Computer Security Foundations Symposium*. Ed. by Steve Chong and Stephanie Delaune. IEEE Computer Society Press, 2018, pp. 119–131. DOI: 10.1109/CSF.2018.00016 (cit. on p. 5).

[Hal05] Shai Halevi. *A plausible approach to computer-aided cryptographic proofs*. Cryptology ePrint Archive, Report 2005/181. https://eprint.iacr.org/2005/181. 2005 (cit. on pp. 4, 182).

[Hv17] Cormac Herley and Paul C. van Oorschot. "SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit". In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 99–120. DOI: 10.1109/SP.2017.38 (cit. on pp. 3, 182, 184).

[Ho+22] S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan. "Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations". In: *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 650–667. DOI: 10.1109/SP46214.2022.00038. URL: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00038 (cit. on pp. 128, 176).

[Hül+21] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. "Post-quantum WireGuard". In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 304–321. DOI: 10.1109/SP40001.2021.00030 (cit. on pp. 183, 185).

[Jag+12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. "On the Security of TLS-DHE in the Standard Model". In: *CRYPTO 2012*. Vol. 7417. LNCS. Springer, 2012, pp. 273–293 (cit. on pp. 7, 61).

[KY05] S. Kent and K. Yao. *Security Architecture for the Internet Protocol*. IETF RFC 4301. 2005 (cit. on p. 34).

[KNB19] N. Kobeissi, G. Nicolas, and K. Bhargavan. "Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 356–370. DOI: 10.1109/EuroSP.2019.00034. URL: https://doi.ieeecomputersociety.org/10.1109/EuroSP.2019.00034 (cit. on pp. 74, 75, 128).

[KBB17] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. “Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach”. In: *2nd IEEE European Symposium on Security and Privacy*. IEEE, Apr. 2017, pp. 435–450 (cit. on pp. 5, 55, 57, 74, 82, 126, 204).

[Kob07] Neal Koblitz. “Another look at automated theorem-proving”. In: *Journal of Mathematical Cryptology* 1.4 (2007), pp. 385–403. DOI: doi:10.1515/jmc.2007.020. URL: https://doi.org/10.1515/jmc.2007.020 (cit. on pp. 182, 184).

[Kob12] Neal Koblitz. “Another look at automated theorem-proving II”. In: *Journal of Mathematical Cryptology* 5.3-4 (2012), pp. 205–224. DOI: doi:10.1515/jmc-2011-0014. URL: https://doi.org/10.1515/jmc-2011-0014 (cit. on p. 182).

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. RFC Editor, Feb. 1997, pp. 1–11. URL: https://www.rfc-editor.org/rfc/rfc2104.html (cit. on p. 107).

[KE10] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. RFC Editor, May 2010, pp. 1–14. URL: https://www.rfc-editor.org/rfc/rfc5869.html (cit. on pp. 41, 56, 107).

[KTG12] Ralf Küsters, Tomasz Truderung, and Juergen Graf. “A Framework for the Cryptographic Verification of Java-Like Programs”. In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 198–212. DOI: 10.1109/CSF.2012.9. URL: https://doi.org/10.1109/CSF.2012.9 (cit. on p. 128).

[LHT16] A. Langley, M. Hamburg, and S. Turner. *Elliptic Curves for Security*. RFC 7748. RFC Editor, Jan. 2016, pp. 1–22. URL: https://www.rfc-editor.org/rfc/rfc7748.html (cit. on pp. 39, 41, 43, 50, 79, 84, 87, 88).

[LGR21] Julia Len, Paul Grubbs, and Thomas Ristenpart. “Partitioning Oracle Attacks”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 195–212. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/len (cit. on p. 119).

[Lip20] Benjamin Lipp. *An Analysis of Hybrid Public Key Encryption*. Cryptology ePrint Archive, Report 2020/243. https://eprint.iacr.org/2020/243. 2020 (cit. on pp. 9, 112).

[LBB19a] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. “A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol”. In: *4th IEEE European Symposium on Security and Privacy*. Full version: https://hal.inria.fr/hal-02100345. Stockholm, Sweden: IEEE Computer Society, June 2019, pp. 231–246. DOI: 10.1109/EuroSP.2019.00026 (cit. on pp. 8, 33, 53).

[LBB19b] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. *A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol*. Research Report RR-9269. Inria Paris, Apr. 2019, p. 49. URL: https://hal.inria.fr/hal-02100345 (cit. on p. 33).

[Low95] Gavin Lowe. “An attack on the Needham-Schroeder public-key authentication protocol”. In: *Information Processing Letters* 56.3 (1995), pp. 131–133. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(95)00144-2. URL: https://www.sciencedirect.com/science/article/pii/0020019095001442 (cit. on pp. 4, 18).

[Low96] Gavin Lowe. “Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 147–166. ISBN: 978-3-540-49874-2 (cit. on p. 4).

[LMN16] Atul Luykx, Bart Mennink, and Samuel Neves. “Security Analysis of BLAKE2s Modes of Operation”. In: *IACR Transactions on Symmetric Cryptology* 2016.1 (Dec. 2016), pp. 158–176 (cit. on p. 43).

[MP16] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. Available at https://signal.org/docs/specifications/x3dh/. Nov. 2016 (cit. on pp. 34, 54).

[Mei+13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Computer Aided Verification, CAV’13*. Vol. 8044. LNCS. Springer, 2013, pp. 696–701 (cit. on pp. 5, 34).

[MF21] Arno Mittelbach and Marc Fischlin. *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography*. Information Security and Cryptography. Springer, 2021. ISBN: 978-3-030-63286-1. DOI: 10.1007/978-3-030-63287-8 (cit. on p. 11).

[Nat13] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. FIPS Publication 186-4. July 2013. URL: https://doi.org/10.6028/nist.fips.186-4 (cit. on pp. 79, 84, 87, 88).

[NS78] Roger M. Needham and Michael D. Schroeder. “Using encryption for authentication in large networks of computers”. In: *Communications of the Association for Computing Machinery* 21.21 (Dec. 1978), pp. 993–999 (cit. on pp. 4, 18).

[NL18] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. IETF RFC 8439. June 2018 (cit. on pp. 41, 43).

[OP01] Tatsuaki Okamoto and David Pointcheval. "The Gap-Problems: a New Class of Problems for the Security of Cryptographic Schemes". In: *PKC 2001*. Vol. 1992. LNCS. Springer, Feb. 2001, pp. 104–118 (cit. on pp. 51, 116).

[Oma+20] E. Omara, Benjamin Beurdouche, E. Rescorla, S. Inguva, A. Kwon, and A. Duric. *The Messaging Layer Security (MLS) Architecture*. Internet-Draft draft-ietf-mls-architecture-05. IETF Secretariat, July 2020. URL: https://tools.ietf.org/html/draft-ietf-mls-architecture-05 (cit. on p. 80).

[Per18] Trevor Perrin. *The Noise Protocol Framework*. https://noiseprotocol.org/noise.html. July 2018 (cit. on pp. 34, 36, 41).

[PM15] Adam Petcher and Greg Morrisett. "The Foundational Cryptography Framework". In: *Principles of Security and Trust*. Ed. by Riccardo Focardi and Andrew Myers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 53–72 (cit. on pp. 6, 128).

[PS10] Alfredo Pironti and Riccardo Sisto. "Provably correct Java implementations of Spi Calculus security protocols specifications". In: *Computers & Security* 29.3 (2010). Special issue on software engineering for secure systems, pp. 302–314. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2009.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0167404809000832 (cit. on p. 128).

[Pol+20] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. "HACLxN: Verified Generic SIMD Crypto (for All Your Favourite Platforms)". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 899918. ISBN: 9781450370899. URL: https://doi.org/10.1145/3372297.3423352 (cit. on pp. 119, 126).

[Pro14] Gordon Procter. *A Security Analysis of the Composition of ChaCha20 and Poly1305*. Cryptology ePrint Archive, Report 2014/613. https://eprint.iacr.org/2014/613. 2014 (cit. on pp. 43, 68).

[Pro+19] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. "Formally Verified Cryptographic Web Applications in WebAssembly". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1256–1274. DOI: 10.1109/SP.2019.00064 (cit. on p. 126).

[Pro+17] Jonathan Protzenko, Jean Karim Zinzindohoue, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Ctlin Hricu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. "Verified Low-Level Programming Embedded in F*". In: *22nd International Conference on Functional Programming (ICFP 2017)*. ACM SIGPLAN, 2017. URL: https://www.microsoft.com/en-us/research/publication/verified-low-level-programming-embedded-f/ (cit. on pp. 126, 167).

[RBS20] Leonie Reichert, Samuel Brack, and Björn Scheuermann. *A Survey of Automatic Contact Tracing Approaches Using Bluetooth Low Energy*. Cryptology ePrint Archive, Report 2020/672. https://eprint.iacr.org/2020/672. 2020 (cit. on p. 3).

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. IETF RFC 8446. 2018 (cit. on pp. 4, 34).

[Res+20] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-07. IETF Secretariat, June 2020. URL: https://tools.ietf.org/html/draft-ietf-tls-esni-07 (cit. on p. 79).

[Rog15] Phillip Rogaway. *The Moral Character of Cryptographic Work*. Cryptology ePrint Archive, Report 2015/1162. https://eprint.iacr.org/2015/1162. 2015 (cit. on pp. 3, 182).

[SA15] M-J. Saarinen and J-P. Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. IETF RFC 7693. 2015 (cit. on p. 41).

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures". In: *ACM CCS 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350 (cit. on p. 185).

[Sho04] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Report 2004/332. https://eprint.iacr.org/2004/332. 2004 (cit. on pp. 4, 11, 13).

[Sis+11] Riccardo Sisto, Matteo Avalle, Alfredo Pironti, and Davide Pozza. "JavaSPI: A Framework for Security Protocol Implementation". In: *Int. J. Secur. Softw. Eng.* 2.4 (2011), pp. 3448. ISSN: 1947-3036. DOI: 10.4018/jsse.2011100103. URL: https://doi.org/10.4018/jsse.2011100103 (cit. on p. 128).

[SD18] A. Suter-Dörig. "Formalizing and verifying the security protocols from the Noise framework". Available at https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/noise_suter-doerig.pdf. Bachelor's thesis. ETH Zürich, Nov. 2018 (cit. on pp. 75, 76).

[Swa+16] Nikhil Swamy, Ctlin Hricu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. "Dependent Types and Multi-Monadic Effects in F*". In: *POPL '16 Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. URL: https://www.microsoft.com/en-us/research/publication/dependent-types-multi-monadic-effects-f/ (cit. on pp. 119, 126).

[Val+22] Henry de Valence, Jack Grigg, Mike Hamburg, Isis Lovecruft, George Tankersley, and Filippo Valsorda. *The ristretto255 and decaf448 Groups*. Internet-Draft draft-irtf-cfrg-ristretto255-decaf448-03. Work in Progress. Internet Engineering Task Force, Feb. 2022. 27 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-03 (cit. on p. 181).

[Zhe97] Yuliang Zheng. "Digital Signcryption or How to Achieve Cost(Signature & Encryption) ≪ Cost(Signature) + Cost(Encryption)". In: *CRYPTO'97*. Ed. by Burton S. Kaliski Jr. Vol. 1294. LNCS. Springer, Heidelberg, Aug. 1997, pp. 165–179. DOI: 10.1007/BFb0052234 (cit. on pp. 79, 83).

Part V

# APPENDIX

# A

# *Appendices to the CryptoVerif Introduction*

## A.1 CryptoVerif Model nsl.ocv

```
(* Needham Schroeder public key protocol, fixed by Lowe.
   The user identity in this model is just an arbitrary
   address. An implementation could use an user@hostname
   construction.
 *)

param Qsetup.
param Qkey_reg.
param Qinit.
param Qresp.

def OptionType_1(option, option_Some, option_None, input) {
  fun option_Some(input): option [data].
  const option_None: option.
  equation forall x: input;
    option_Some(x) <> option_None.
}

type nonce [fixed,large].
type pkey [bounded].
type skey [bounded].
type keyseed [fixed,large].
type encseed [bounded].
type plaintext [bounded].
type address [bounded].
type keypair [bounded].
fun kp(pkey, skey): keypair [data].

type ciphertext [bounded].
type ciphertext_opt [bounded].
expand OptionType_1(ciphertext_opt, ciphertext_some, ciphertext_bottom, ciphertext).

table trusted_keys(address, skey, pkey).
table all_keys(address, pkey, bool).

set diffConstants = false.

fun msg1(nonce, address):plaintext [data].
fun msg2(nonce, nonce, address):plaintext [data].
fun msg3(nonce):plaintext [data].


```

```
equation forall z:nonce,t:nonce,u:address,y2:nonce,z2:address;
  msg2(z,t,u) <> msg1(y2,z2).
equation forall y:nonce,y2:nonce,z2:address;
  msg3(y) <> msg1(y2,z2).
equation forall z:nonce,t:nonce,u:address,y2:nonce;
  msg2(z,t,u) <> msg3(y2).

(* Public-key encryption (IND-CCA2) *)

proba Penc.
proba Penccoll.

expand IND_CCA2_public_key_enc_all_args(
  keyseed, pkey, skey, plaintext, ciphertext_opt, encseed,
  skgen, skgen2, pkgen, pkgen2, enc, enc_r, enc_r2, dec_opt, dec_opt2, injbot,
  Z, Penc, Penccoll).

(* Not needed because the in processes receive ciphertext, not ciphertext_opt *)
equation forall sk: skey; dec_opt(ciphertext_bottom, sk) = bottom.

letfun dec(c: ciphertext, sk: skey) =
  dec_opt(ciphertext_some(c), sk).

letfun keygen() = k <-R keyseed; kp(pkgen(k), skgen(k)).

const Zplaintext: plaintext.
equation forall x: plaintext; Z(x) = Zplaintext.

(* Queries *)

event beginA(address, address, nonce, nonce).
event endA(address, address, nonce, nonce).
event beginB(address, address, nonce, nonce).
event endB(address, address, nonce, nonce).

query x:address, y:address, na:nonce, nb:nonce;
  event(endA(x,y,na,nb)) ==> event(beginB(x,y,na,nb)).
query x:address, y:address, na:nonce, nb:nonce;
  event(endB(x,y,na,nb)) ==> event(beginA(x,y,na,nb)).
query x:address, y:address, na:nonce, nb:nonce;
  inj-event(endA(x,y,na,nb)) ==> inj-event(beginB(x,y,na,nb)).
query x:address, y:address, na:nonce, nb:nonce;
  inj-event(endB(x,y,na,nb)) ==> inj-event(beginA(x,y,na,nb)).

let initiator() =

  Initiator {

    foreach i_init ≤ Qinit do

    initiator_send_msg1(addrA: address, addrX: address) :=
      (* the gets fail if addrA or addrX have not been
         setup by the adversary. *)
      get[unique] trusted_keys(=addrA, skA, pkA) in
      get[unique] all_keys(=addrX, pkX, trustX) in
      (* Prepare Message 1 *)
      Na <-R nonce;
      let cc1 = enc(msg1(Na, addrA), pkX) in
      let ciphertext_some(c1: ciphertext) = cc1 in
      return (c1);

    (* Receive Message 2 *)
```

```
106    initiator_send_msg3 (c: ciphertext) :=
107      let injbot(msg2(=Na, Nb, =addrX)) = dec(c, skA) in
108      event beginA(addrA, addrX, Na, Nb);
109      (* Prepare Message 3 *)
110      let ciphertext_some(c3) = enc(msg3(Nb), pkX) in
111      return (c3);
112
113    (* OK *)
114    initiator_finish () :=
115      if (trustX) then
116      event endA(addrA, addrX, Na, Nb);
117      return ()
118
119  }.
120
121 let responder() =
122
123  Responder {
124
125  foreach i_resp ≤ Qresp do
126
127  (* Receive Message 1 *)
128  responder_send_msg2 (addrB: address, m: ciphertext) :=
129    (* the get fails if addrB has not been setup by
130       the adversary *)
131    get[unique] trusted_keys(=addrB, skB, pkB) in
132    let injbot(msg1(Na, addrY)) = dec(m, skB) in
133    get[unique] all_keys(=addrY, pkY, trustY) in
134    (* Send Message 2 *)
135    Nb <-R nonce;
136    event beginB(addrY, addrB, Na, Nb);
137    let ciphertext_some(c2) = enc(msg2(Na, Nb, addrB), pkY) in
138    return (c2);
139
140    (* Receive Message 3 *)
141    responder_receive_msg3 (m3: ciphertext) :=
142      let injbot(msg3(=Nb)) = dec(m3, skB) in
143      if (trustY) then (
144        event endB(addrY, addrB, Na, Nb); return ())
145      else return ()
146
147  }.
148
149 let key_register() =
150  Key_Register {
151
152  foreach i ≤ Qkey_reg do
153
154  register (addr: address, pkX: pkey) :=
155    get[unique] all_keys(=addr, ign1, ign2) in (
156      yield
157    ) else
158      insert all_keys(addr, pkX, false);
159      return ()
160  }.
161
162 let setup() =
163  Setup {
164
165    foreach i ≤ Qsetup do
166      setup(addr: address) :=
167        get[unique] all_keys(=addr, ign1, ign2) in (
```

```
168         yield
169       ) else
170         let kp(the_pkA: pkey, the_skA: skey) = keygen() in
171         insert trusted_keys(addr, the_skA, the_pkA);
172         insert all_keys(addr, the_pkA, true);
173         return(the_pkA)
174 
175   }.
176 
177 process
178   (
179     run setup()
180   |
181     run key_register()
182   |
183     run initiator()
184   |
185     run responder()
186   )
```

# B

# *Appendices to the WireGuard Analysis*

## B.1 Indifferentiability Results

### B.1.1 *Basic Lemmas*

*Proof of Lemma 3.2.* Consider

- the game $G_0$ in which $H$ is a random oracle, and $H_i(x) = H(x)$ for each $x \in D_i$ and $i \leq n$, and
- the game $G_1$ in which $H_1, \ldots H_n$ are independent random oracles defined on $D_1, \ldots, D_n$ respectively, and $H(x) = H_i(x)$ if $x \in D_i$ for some $i \leq n$, and $H(x) = H_0(x)$ otherwise, where $H_0$ is a random oracle of domain $D \setminus (D_1 \cup \cdots \cup D_n)$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. □

*Proof of Lemma 3.3.* Consider

- the game $G_0$ in which $H_1$ and $H_2$ are independent random oracles, and $H'(x) = H_1(x) \| H_2(x)$, and
- the game $G_1$ in which $H'$ is a random oracle that returns bitstrings of length $l_1 + l_2$, $H_1(x)$ is the $l_1$ first bits of $H'(x)$ and $H_2(x)$ is the $l_2$ last bits of $H'(x)$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. □

*Proof of Lemma 3.4.* Consider

- the game $G_0$ in which $H$ is a random oracle, $H'_1(x)$ is the first $l_1$ bits of $H(x)$, and $H'_2(x)$ is the last $l - l_1$ bits of $H(x)$, and
- the game $G_1$ in which $H'_1$ and $H'_2$ are independent random oracles that return bitstrings of length $l_1$ and $l - l_1$ respectively, and $H(x) = H'_1(x) \| H'_2(x)$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. (It is the same indistinguishability result as in Lemma 3.3, swapping $G_0$ and $G_1$.) □

*Proof of Lemma 3.5.* This is a consequence of Lemma 3.4, by not giving access to oracle $H'_2$ to the distinguisher (so $q_{H'_2} = 0$). $H'_2$ is then included in the simulator. We assume that random oracles answer in constant time. □

*Proof of Lemma 3.7.* Consider

- the game $G_0$ in which $H_1$ and $H_2$ are independent random oracles, $H'_1(x) = H_1(x)$, and $H'_2(x) = H_2(H_1(x), x)$, and
- the game $G_1$ in which $H'_1$ and $H'_2$ are independent random oracles; $H_1(x) = H'_1(x)$; $H_2(y,z)$ returns $H'_2(z)$ if $y = H'_1(z)$ and $H_3(y,z)$ otherwise, where $H_3$ is a random oracle (independent of $H'_1$ and $H'_2$).

CryptoVerif shows that these two games are indistinguishable, up to probability $\epsilon$. (The oracles $H_1$ and $H'_1$ are considered as a single oracle, which receives $q_{H_1} + q_{H'_1}$ queries in total.) □

### B.1.2 *Indifferentiability of HKDF*

Much like for HMAC in [Dod+12] and as mentioned in [KBB17], $\mathsf{hkdf}_n$ is not indifferentiable from a random oracle in general. Intuitively, the problem comes from a confusion between the first and the second (or third) call to hmac, which makes it possible to generate PRK by calling $\mathsf{hkdf}_2$ rather than hmac. In more detail, let

[Dod+12] Dodis et al., "To Hash or Not to Hash Again? (In)Differentiability Results for $H^2$ and HMAC"

[KBB17] Kobeissi et al., "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach"

$$\begin{aligned}
&\mathrm{PRK}\|_ = \mathsf{hkdf}_2(s, k, \mathrm{info})\\
&\mathrm{salt} = \mathsf{hmac}(s, k)\\
&x = \mathsf{hmac}(\mathrm{PRK}, \mathrm{info}'\|i_0)\\
&x'\|_ = \mathsf{hkdf}_2(\mathrm{salt}, \mathrm{info}\|i_0, \mathrm{info}')
\end{aligned}$$

where the notation $x_1\|x_2 = \mathsf{hkdf}_2(s, k, \mathrm{info})$ denotes that $x_1$ consists of the first 256 bits of $\mathsf{hkdf}_2(s, k, \mathrm{info})$ and $x_2$ its last 256 bits.

When $\mathsf{hkdf}_2$ is defined from hmac as above, we have $\mathrm{PRK} = \mathsf{hmac}(\mathrm{PRK}', \mathrm{info}\|i_0)$ where $\mathrm{PRK}' = \mathsf{hmac}(s, k) = \mathrm{salt}$, so $\mathrm{PRK} = \mathsf{hmac}(\mathrm{salt}, \mathrm{info}\|i_0)$. Hence, $x' = \mathsf{hmac}(\mathrm{PRK}, \mathrm{info}'\|i_0) = x$. However, when $\mathsf{hkdf}_2$ is a random oracle and hmac is defined from $\mathsf{hkdf}_2$, the simulator that computes hmac sees what seems to be two unrelated calls to hmac. (It is unable to see that PRK is in fact related to the previous call $\mathrm{salt} = \mathsf{hmac}(s, k)$: we have $\mathrm{PRK}\|\_ = \mathsf{hkdf}_2(s, k, \mathrm{info})$ but the simulator does not know which value of info it should use.) Therefore, the simulator can only return fresh random values for salt and $x$, and $x \neq x'$ in general.

*Proof of Lemma 3.8.* In this proof, we write $S[H_1, \ldots, H_n]$ instead of $S^{H_1,\ldots,H_n}$ for a system $S$ with oracle access to $H_1, \ldots, H_n$, because we need to write systems in which the oracles are themselves systems that access other oracles. Consider the game $G_0$ in which hmac is a random oracle and $\mathsf{hkdf\text{-}expand}_n$ is defined as above.

The different calls to hmac in the definition of $\mathsf{hkdf\text{-}expand}_n$ use disjoint domains (the last byte differs among the calls to hmac), so by Lemma 3.2, there exists a simulator $S_1$ for hmac such that $G_0$ is perfectly indistinguishable from $G_1$ in which $\mathsf{hmac} = S_1[H_1 \ldots, H_n]$ and $\mathsf{hkdf\text{-}expand}_n$ is defined by

$$\begin{aligned}
&\mathsf{hkdf\text{-}expand}_n^1(\mathrm{PRK}, \mathrm{info}) = k_1\| \ldots \|k_n \text{ where}\\
&\quad k_1 = H_1(\mathrm{PRK}, \mathrm{info})\\
&\quad k_{i+1} = H_{i+1}(\mathrm{PRK}, k_i\|\mathrm{info}) \text{ for } 1 \leq i < n
\end{aligned}$$

where $H_1, \ldots, H_n$ are independent random oracles and the simulator $S_1$ calls $H_i$ at most $q_{H_i}$ times, with $q_{H_1} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$.

Slightly reorganising the arguments of $H_i$, there exists a simulator $S_2$ for hmac such that $G_1$ is perfectly indistinguishable from $G_2$ in which $\mathsf{hmac} = S_2[H_1, \ldots, H_n]$ and $\mathsf{hkdf\text{-}expand}_n$ is defined by

$$\begin{aligned}&\mathsf{hkdf\text{-}expand}_n^2(\mathrm{PRK}, \mathrm{info}) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad k_1 = H_1(\mathrm{PRK}, \mathrm{info})\\ &\quad k_{i+1} = H_{i+1}(k_i, \mathrm{PRK}, \mathrm{info}) \text{ for } 1 \leq i < n\end{aligned}$$

where $H_1, \ldots, H_n$ are independent random oracles and the simulator $S_2$ calls $H_i$ at most $q_{H_i}$ times, with $q_{H_1} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$.

By Lemma 3.7, there exists a simulator $S_{3,2}$ for $H_2$ ($H_1$ is simulated by $H'_1 = H_1$ itself) such that $G_2 = G_{3,1}$ is indistinguishable up to probability $\epsilon_2$ from $G_{3,2}$ in which $\mathsf{hmac} = S_2[H'_1, S_{3,2}[H'_1, H'_2], H_3, \ldots, H_n]$ and $\mathsf{hkdf\text{-}expand}_n$ is defined by

$$\begin{aligned}&\mathsf{hkdf\text{-}expand}_n^{3,2}(\mathrm{PRK}, \mathrm{info}) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad k_1 = H'_1(\mathrm{PRK}, \mathrm{info})\\ &\quad k_2 = H'_2(\mathrm{PRK}, \mathrm{info})\\ &\quad k_{i+1} = H_{i+1}(k_i, \mathrm{PRK}, \mathrm{info}) \text{ for } 2 \leq i < n\end{aligned}$$

where $H'_1, H'_2, H_3, \ldots, H_n$ are independent random oracles; the simulator for hmac calls $H'_1$ at most $q_{H_1} + q_{H_2}$ times, $H'_2$ at most $q_{H_2}$ times, $H_i$ at most $q_{H_i}$ times for $i \geq 3$, with $q_{H_1} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}} + q_{H_2})$; and $\epsilon_2 = q_{H_2}(2q_{H_1} + 3q_{\mathsf{hkdf\text{-}expand}_n} + 1)/|\mathcal{M}|$. Furthermore, by definition of the simulator in the proof of Lemma 3.7, $H'_1$ is called at most $q_{H_1}$ times via the first hole of $S_2$ and $H'_1$ and $H'_2$ are called with the same arguments at most $q_{H_2}$ times via the second hole of $S_2$.

Repeating the same reasoning inductively, there exists a simulator $S_{3,j}$ for $H_j$ such that $G_{3,j-1}$ is indistinguishable up to probability $\epsilon_j$ from $G_{3,j}$ in which $\mathsf{hmac} = S_2[H'_1, S_{3,2}[H'_1, H'_2], \ldots, S_{3,j}[H'_{j-1}, H'_j], H_{j+1}, \ldots, H_n]$ and $\mathsf{hkdf\text{-}expand}_n$ is defined by

$$\begin{aligned}&\mathsf{hkdf\text{-}expand}_n^{3,j}(\mathrm{PRK}, \mathrm{info}) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad k_i = H'_i(\mathrm{PRK}, \mathrm{info}) \text{ for } 1 \leq i \leq j\\ &\quad k_{i+1} = H_{i+1}(k_i, \mathrm{PRK}, \mathrm{info}) \text{ for } j \leq i < n\end{aligned}$$

where $H'_1, \ldots, H'_j, H_{j+1}, \ldots, H_n$ are independent random oracles; the simulator for hmac calls $H'_1$ at most $q_{H_1}$ times via the first hole of $S_2$, $H'_{i-1}$ and $H'_i$ with the same arguments at most $q_{H_i}$ times via the $i$-th hole of $S_2$ for $1 < i \leq j$, $H_i$ at most $q_{H_i}$ times for $i > j$, with $q_{H_0} + \cdots + q_{H_n} \leq q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}} + q_{H_2} + \cdots + q_{H_j})$; and $\epsilon_j = q_{H_j}(2q_{H_{j-1}} + 3q_{\mathsf{hkdf\text{-}expand}_n} + 1)/|\mathcal{M}|$.

For $j = n$, we obtain a game $G_{3,n}$ in which $\mathsf{hmac} = S_2[H'_1, S_{3,2}[H'_1, H'_2], \ldots, S_{3,n}[H'_{n-1}, H'_n]]$ and $\mathsf{hkdf\text{-}expand}_n$ is defined by

$$\begin{aligned}&\mathsf{hkdf\text{-}expand}_n^{3,n}(\mathrm{PRK}, \mathrm{info}) = k_1 \| \ldots \| k_n \text{ where}\\ &\quad k_i = H'_i(\mathrm{PRK}, \mathrm{info}) \text{ for } 1 \leq i \leq n\end{aligned}$$

where $H'_1, \ldots, H'_n$ are independent random oracles; the simulator for hmac calls $H'_1$ at most $q_{H_1}$ times via the first hole of $S_2$, $H'_{i-1}$ and $H'_i$ with the

same arguments at most $q_{H_i}$ times via the $i$-th hole of $S_2$ for $1 < i \le n$, with $q_{H_1}+\cdots+q_{H_n} \le q_{\mathsf{hmac}}$, and runs in time $\mathcal{O}(q_{\mathsf{hmac}}+q_{H_2}+\cdots+q_{H_n}) = \mathcal{O}(q_{\mathsf{hmac}})$.

By Lemma 3.3, there exist simulators $S_{4,j}$ $(1 \le j \le n)$ for $H'_j$ such that $G_{3,n}$ is perfectly indistinguishable from $G_4$ in which $\mathsf{hmac} = S_2[S_{4,1}[H], S_{3,2}[S_{4,1}[H], S_{4,2}[H]], \ldots, S_{3,n}[S_{4,n-1}[H], S_{4,n}[H]]]$ and $\mathsf{hkdf\text{-}expand}_n$ is a random oracle $H$ where the simulator for $\mathsf{hmac}$ calls $H$ at most $q_{H_1}+\cdots+q_{H_n} \le q_{\mathsf{hmac}}$ times, and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$. (Since $H$ is the concatenation of the $H'_i$ for $1 \le i \le n$ and, for $i > 1$, the calls to $H'_{i-1}$ and $H'_i$ via the $i$-th hole of $S_2$ have the same arguments, these two calls can be implemented by a single call to $H$, and hence there are at most $q_{H_i}$ calls to $H$ via the $i$-th hole of $S_2$.)

The probability of distinguishing $G_0$ from $G_4$ is then at most

$$\begin{aligned}
\epsilon &= \sum_{j=2}^{n} \epsilon_j \\
&= \sum_{j=2}^{n} \frac{q_{H_j}(2q_{H_{j-1}} + 3q_{\mathsf{hkdf\text{-}expand}_n} + 1)}{|\mathcal{M}|} \\
&= \frac{1}{|\mathcal{M}|}\left(3q_{\mathsf{hkdf\text{-}expand}_n}(\sum_{j=2}^{n} q_{H_j}) + 2\sum_{j=2}^{n} q_{H_j} q_{H_{j-1}} + \sum_{j=2}^{n} q_{H_j}\right) \\
&\le \frac{1}{|\mathcal{M}|}\left(3q_{\mathsf{hkdf\text{-}expand}_n} q_{\mathsf{hmac}} + 2\sum_{j=2}^{n} q_{H_j} q_{H_{j-1}} + \sum_{j=2}^{n} q_{H_j}^2\right) \\
&\le \frac{3q_{\mathsf{hkdf\text{-}expand}_n} q_{\mathsf{hmac}} + q_{\mathsf{hmac}}^2}{|\mathcal{M}|}
\end{aligned}$$

□

□

*Proof of Lemma 3.9.* As in the previous proof, we write $S[H_1, \ldots, H_n]$ instead of $S^{H_1,\ldots,H_n}$ for a system $S$ with oracle access to $H_1, \ldots, H_n$. Consider the game $G_0$ in which $\mathsf{hmac}$ is a random oracle and $\mathsf{hkdf}_n$ is defined as above.

By hypothesis, the calls to $\mathsf{hmac}$ in $\mathsf{hkdf\text{-}extract}$ and $\mathsf{hkdf\text{-}expand}_n$ use disjoint domains, so by Lemma 3.2, there exists a simulator $S_1$ for $\mathsf{hmac}$ such that $G_0$ is perfectly indistinguishable from $G_1$ in which $\mathsf{hmac} = S_1[H_0, H_1]$ and $\mathsf{hkdf}_n$ is defined by $\mathsf{hkdf}_n^1(\mathrm{salt}, k, \mathrm{info}) = \mathsf{hkdf\text{-}expand}_n^1(H_0(\mathrm{salt}, k), \mathrm{info})$, where $\mathsf{hkdf\text{-}expand}_n^1$ is defined from the random oracle $H_1$ instead of $\mathsf{hmac}$ and the simulator $S_1$ calls $H_i$ at most $q_{H_i}$ times, with $q_{H_0} + q_{H_1} \le q_{\mathsf{hmac}}$ and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$.

By Lemma 3.8, there exists a simulator $S_2$ for $H_1$ such that $G_1$ is indistinguishable up to probability $\epsilon$ from $G_2$ in which $\mathsf{hmac} = S_1[H_0, S_2[H]]$, and $\mathsf{hkdf}_n$ is defined by $\mathsf{hkdf}_n^1(\mathrm{salt}, k, \mathrm{info}) = H(H_0(\mathrm{salt}, k), \mathrm{info})$, where $H$ is a random oracle, $S_2$ calls $H$ at most $q_H \le q_{H_1}$ times, runs in time $\mathcal{O}(q_{H_1}) = \mathcal{O}(q_{\mathsf{hmac}})$, and $\epsilon = (3q_{\mathsf{hkdf}_n} q_{H_1} + q_{H_1}^2)/|\mathcal{M}|$. So the simulator for $\mathsf{hmac}$ calls $H_0$ at most $q_{H_0}$ times and $H$ at most $q_H \le q_{H_1}$ times and runs in time $\mathcal{O}(q_{\mathsf{hmac}})$.

By Lemma 3.6, there exist simulators $S_{3,1}$ for $H_0$ and $S_{3,2}$ for $H$ such that $G_2$ is indistinguishable up to probability $\epsilon'$ from $G_3$ in which $\mathsf{hmac} = S_2[H_0, S_2[H]]]$ with $H_0 = S_{3,1}[\mathsf{hkdf}_n^3]$ and $H = S_{3,2}[\mathsf{hkdf}_n^3]$ and $\mathsf{hkdf}_n = \mathsf{hkdf}_n^3$ is a random oracle, where the simulator for $\mathsf{hmac}$ calls $\mathsf{hkdf}_n$ at most $q_H \le q_{H_1} \le q_{\mathsf{hmac}}$ times, runs in time $\mathcal{O}(q_{\mathsf{hmac}} + q_{H_0} q_H) = \mathcal{O}(q_{\mathsf{hmac}}^2)$, and $\epsilon' = (2q_H q_{H_0} + q_{H_0}^2 + q_H q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2)/|\mathcal{M}|$.

The probability of distinguishing $G_0$ from $G_3$ is then at most

$$\begin{aligned}
\epsilon &= \epsilon' + \epsilon \\
&= \frac{2q_H q_{H_0} + q_{H_0}^2 + q_H q_{\mathsf{hkdf}_n} + q_{\mathsf{hkdf}_n}^2}{|\mathcal{M}|} + \frac{3q_{\mathsf{hkdf}_n} q_{H_1} + q_{H_1}^2}{|\mathcal{M}|} \\
&= \frac{1}{|\mathcal{M}|}\left(q_{\mathsf{hkdf}_n}^2 + q_{\mathsf{hkdf}_n}(q_H + 3q_{H_1}) + q_{H_1}^2 + 2q_H q_{H_0} + q_{H_0}^2\right) \\
&\leq \frac{q_{\mathsf{hkdf}_n}^2 + 4q_{\mathsf{hkdf}_n} q_{\mathsf{hmac}} + q_{\mathsf{hmac}}^2}{|\mathcal{M}|}
\end{aligned}$$

□

□

### B.1.3 *Chain of Random Oracle Calls*

*Proof of Lemma 3.10.* We consider the following two games $G_0$ and $G_1$.

- The game $G_0$ in which $\mathsf{H}$ is a random oracle and the functions $\mathsf{chain}_n$ with $0 \leq n \leq m$ are defined from $\mathsf{H}$ by (3.11) and (3.12).
- The game $G_1$ in which the functions $\mathsf{chain}_n$ with $0 \leq n \leq m$ are independent random oracles and $\mathsf{H}$ is defined from them in Figure B.1. In this figure, $L$ is a list of triples $((C, v), (C_{j+1}, r_j), j)$ such that $C_{j+1} \| r_j$ is the result of a previous call to $\mathsf{H}(C, v)$ and $j$ indicates the index of this $\mathsf{H}$ call in a chain of calls to $\mathsf{H}$. If a call to $\mathsf{H}$ was not coming from a chain of calls, the index $j = -2$ is used.

  We shortly comment this simulator's four cases informally. In case 1, it returns a previous result because the same call has already been made before. In case 2, the call to $\mathsf{H}$ uses const as first argument and is thus the first call in a potential chain of calls to $\mathsf{H}$. Therefore, the simulator uses $\mathsf{chain}_0$ to get the result and writes it to the list $L$ with index 0. In case 3, the simulator finds in $L$ a previous call to $\mathsf{H}$ that returned the current call's $C$ value as result. This means, with respect to the hypothesis we present just after this paragraph, that the current call belongs to a chain of previous calls that was started with a call responded to by case 2. The simulator collects the arguments $v_k$ of those previous calls to be able to call the appropriate $\mathsf{chain}_n$ oracle. If the simulator reaches case 4, then the call did neither start a new chain nor belong to a previously started chain. Thus, it chooses fresh random values as a result and adds them with an index to $L$ that makes sure that it will never be considered as part of a chain.

We name *direct* oracle calls to $\mathsf{chain}_n$ or $\mathsf{H}$ calls that are done directly by the distinguisher, and *indirect* oracle calls the calls to $\mathsf{H}$ done from inside $\mathsf{chain}_n$ in $G_0$ and the calls to $\mathsf{chain}_n$ done from inside $\mathsf{H}$ in $G_1$. Note for clarification that in $G_0$ there are no indirect calls to $\mathsf{chain}_n$ and in $G_1$ there are no indirect calls to $\mathsf{H}$.

We show that the two games $G_0$ and $G_1$ are indistinguishable as long as the following hypotheses hold. In game $G_0$:

H1. Consider $C_{j+1} \| r_j = \mathsf{H}(C_j, v_j), j \leq n < m$ that gets called from inside a direct call to $\mathsf{chain}_n(v_1, \ldots, v_j, \ldots, v_n)$. If the distinguisher calls $\mathsf{H}(C_{j+1}, v_{j+1})$ before or after the call to $\mathsf{chain}_n$, then $\mathsf{H}(C_j, v_j)$ has been called directly by the distinguisher before $\mathsf{H}(C_{j+1}, v_{j+1})$.

FIGURE B.1: Simulator for $\mathsf{H}$

$\mathsf{H}(C, v) =$

1) **if** $((C, v), (C_{j+1}, r_j), j) \in L$ for some $C_{j+1}, r_j, j$ **then**
   **return** $C_{j+1} \| r_j$
2) **else if** $C = \mathsf{const}$ **then**
   $C_1 \leftarrow\!\!\$\ \{0,1\}^{l'}$
   $r_0 \leftarrow \mathsf{chain}_0(v)$
   add $((C, v), (C_1, r_0), 0)$ to $L$
   **return** $C_1 \| r_0$
3) **else if** $((C_j, v_j), (C, r_j), j) \in L$ for some $C_j, v_j, r_j$ and $j$ with $0 \le j < m$ **then**
   **for** $k = j - 1$ to 0 **do**
     find $((C_k, v_k), (C_{k+1}, r_k), k) \in L$ for some $C_k, v_k, r_k$
     set $C_k$ and $v_k$ to the found values
   endfor
   **if** $j + 1 = m$ **then**
     $C_{j+2} \| r_{j+1} = \mathsf{chain}_m(v_0, \ldots, v_j, v)$
   **else**
     $C_{j+2} \leftarrow\!\!\$\ \{0,1\}^{l'}$
     $r_{j+1} \leftarrow \mathsf{chain}_{j+1}(v_0, \ldots, v_j, v)$
   **endif**
   add $((C, v), (C_{j+2}, r_{j+1}), j + 1)$ to $L$
   **return** $C_{j+2} \| r_{j+1}$
4) **else**
   $C_{-1} \leftarrow\!\!\$\ \{0,1\}^{l'}$
   $r_{-2} \leftarrow\!\!\$\ \{0,1\}^{l-l'}$
   add $((C, v), (C_{-1}, r_{-2}), -2)$ to $L$
   **return** $C_{-1} \| r_{-2}$
  **endif**

Stated informally, the distinguisher can only know $C_{j+1}$ if it was received as result from $\mathsf{H}$.

In game $G_1$:

H2. No fresh $C_j$ is equal to the first argument of a previous call to $\mathsf{H}$ (including the call that generates $C_j$).

Stated informally, the distinguisher cannot prepend to a chain of $\mathsf{H}$ calls.

H3. There are no collisions between the fresh $C_j$.

We have the following invariants:

P1. Given $C, v$, there is at most one pair $((C_{j+1}, r_j), j)$ such that

$$((C, v), (C_{j+1}, r_j), j) \in L\,.$$

Indeed, when $L$ contains such an element, calls to $\mathsf{H}(C, v)$ immediately return $C_{j+1} \| r_j$ in case 1, and never add another element $((C, v), (C_{j+1}, r_j), j)$ to $L$.

P2. Given $((C_j, v_j), (C, r_j), j) \in L$, for some $C_j, v_j, r_j$ and $j$ with $0 \leq j < m$ then there is, for each $k = j - 1$ to 0, a single matching element $((C_k, v_k), (C_{k+1}, r_k), k)$ in $L$.

More informally, at no time there are entries in $L$ that belong to chains that are incomplete in the front, i. e. that did not start by a call to $\mathsf{H}$ with $C = \mathsf{const}$. And yet differently stated, the simulator can, in case 3, always reconstruct the whole chain of $\mathsf{H}$ calls and collect the arguments $v_j$.

We first show the existence of $((C_k, v_k), (C_{k+1}, r_k), k)$ in $L$ for each $k = j - 1$ to 0. If there is an element in $L$ with $j > 0$, then case 3 was executed before for a matching $\mathsf{H}$ call and its result was added to the list with $j' = j - 1$. This is because of H2 and the fact that only in case 3 elements with $j > 0$ are added to the list. This argument can be repeated recursively until reaching $j = 1$. For $j = 0$, the matching element that started the chain was added by case 2, once again because of H2 and because only in case 2 elements with $j = 0$ are added to the list.

Moreover, the uniqueness of $((C_k, v_k), (C_{k+1}, r_k), k)$ comes from H3: when an element $((C_k, v_k), (C_{k+1}, r_k), k)$ is added to $L$, $C_{k+1}$ is always a fresh $C_j$, so by H3, there is a single element in $L$ with a given $C_{k+1}$.

We now treat all possible traces of calls in both games.

CASE 1 Suppose the distinguisher makes a direct oracle call to $\mathsf{H}$ or $\mathsf{chain}_n$ with the same arguments as a previous direct call to the same oracle. Both $G_0$ and $G_1$ return the same result as in the previous call.

CASE 2 Suppose the distinguisher makes a direct call to $\mathsf{chain}_n$ that has not been done before as a direct call.

CASE 2. A) In $G_0$, the last $\mathsf{H}(C, v_n)$ in the chain that simulates $\mathsf{chain}_n(v_0, \ldots, v_n)$ *has* already been called directly. Then by H1 the distinguisher did all $\mathsf{H}$ calls in the chain that simulates $\mathsf{chain}_n(v_0, \ldots, v_n)$ directly.

The result in $G_0$ is

$$\_ \| \mathsf{chain}_n(v_0, \ldots, v_n) = \mathsf{H}(C, v_n)$$

which is the last part of the result of the previous call to $\mathsf{H}$, or in the case of $n = m$

$$\mathsf{chain}_m(v_0, \ldots, v_m) = \mathsf{H}(C, v_m)\,.$$

In $G_1$, because the whole chain of $\mathsf{H}$ calls was made in the right order, $\mathsf{chain}_n(v_0, \ldots, v_n)$ has already been invoked indirectly by the call to $\mathsf{H}(C, v_n)$. Thus, this current call to $\mathsf{chain}_n$ returns a previously fixed value, fulfilling the following equation:

$$\mathsf{H}(C, v_n) = C_{j+1} \| \mathsf{chain}_n(v_0, \ldots, v_n)$$

or in the case of $n = m$

$$\mathsf{H}(C, v_m) = \mathsf{chain}_m(v_0, \ldots, v_m)\,.$$

This is the same result as in $G_0$.

CASE 2. B) In $G_0$, the last $\mathsf{H}(C, v_n)$ in the chain that simulates $\mathsf{chain}_n(v_0, \ldots, v_n)$ *has not* already been called directly. Like in the previous case, the result is

$$\_ \| \mathsf{chain}_n(v_0, \ldots, v_n) = \mathsf{H}(C, v_n)$$

or in the case of $n = m$

$$\mathsf{chain}_m(v_0, \ldots, v_m) = \mathsf{H}(C, v_m)\,,$$

but as $\mathsf{H}(C, v_n)$ and $\mathsf{chain}_n(v_1, \ldots, v_n)$ have not been called before directly, the result is independent of previously returned values and thus looks like a fresh random value to the distinguisher.

In $G_1$, $\mathsf{chain}_n(v_0, \ldots, v_n)$ has not been invoked before and thus returns a fresh random value.

CASE 3 Suppose the distinguisher makes a direct call to $\mathsf{H}$ that has not been done before as a direct call.

CASE 3. A) In $G_0$, this call to $\mathsf{H}(C, v_i)$ *has* already been done from inside a $\mathsf{chain}_n(v_0, \ldots, v_n)$ call. Hence all other $\mathsf{H}$ calls belonging to this chain have also been done from inside said $\mathsf{chain}_n$ call, in particular the call $\mathsf{H}(C_{i-1}, v_{i-1})$ directly before the current call (except for $C = \mathsf{const}$, thus if the current call is the beginning of a chain). H1 implies that the distinguisher has then made a direct call to $\mathsf{H}(C_{i-1}, v_{i-1})$ before the current $\mathsf{H}$ call. By recursively applying H1, the distinguisher has then directly made all $\mathsf{H}$ calls in the chain up to the current one, in the right order.

CASE 3. A) I) In $G_0$, the current direct call to $\mathsf{H}(C, v_n)$ has already been done as *the last one* of the chain of calls indirectly invoked from inside a $\mathsf{chain}_n(v_0, \ldots, v_n)$ call.

In $G_0$, the result fulfils the following equation:

$$C_{n+1} \| \mathsf{chain}_n(v_0, \ldots, v_n) = \mathsf{H}(C, v_n)\,,$$

and in the case of $n = m$:

$$\mathsf{chain}_m(v_0, \ldots, v_m) = \mathsf{H}(C, v_m)\,.$$

This is similar to case 2. a) just that the order of the calls is inverted and the following small difference: the parts of $\mathsf{H}$'s result coming from $\mathsf{chain}$ are already known by the distinguisher, while $C_{n+1}$ looks like a fresh random value.

In $G_1$, because the whole chain of $\mathsf{H}$ calls was made in the right order, the current call will invoke case 3 of the simulator's algorithm and return

$$\mathsf{H}(C, v_n) = C_{n+1} \| \mathsf{chain}_n(v_0, \ldots, v_n)$$

or in the case of $n = m$

$$\mathsf{H}(C, v_m) = \mathsf{chain}_m(v_0, \ldots, v_m)\,.$$

The parts of $\mathsf{H}$'s result coming from $\mathsf{chain}$ are already known by the distinguisher, while $C_{n+1}$ is a fresh random value. This is indistinguishable from the result in $G_0$.

CASE 3. A) II) In $G_0$, the current direct call to $\mathsf{H}(C, v_i)$ has already been done from inside a $\mathsf{chain}_n(v_0, \ldots, v_n)$ call, but *not as the last one*. This implies that said $\mathsf{chain}_n$ call was not $\mathsf{chain}_0$ – this is covered by case 3. a) i).

In $G_0$, the result is thus a value fixed by a previous indirect call to $\mathsf{H}$, but is independent of the results of previous direct calls, and thus looks like a fresh random value to the distinguisher.

In $G_1$, because the whole chain of $\mathsf{H}$ calls was made in the right order, the current call will invoke case 3 of the simulator's algorithm and return a result via a $\mathsf{chain}_n$ call. This $\mathsf{chain}_n$ call has not been made before by hypothesis and thus the result is a fresh random value.

CASE 3. B) In $G_0$, this call to $\mathsf{H}(C, v_i)$ *has not* been done before, neither directly nor indirectly.[1] Hence, $\mathsf{H}$ returns a fresh random value.

[1] This means that there is no involvement of previous calls to $\mathsf{chain}_n$, but the distinguisher can build an $\mathsf{H}$ chain with direct calls.

In $G_1$, the simulator's case 1 is not relevant because this call has not been done before. Simulator's case 2: If $C = \mathsf{const}$, then $\mathsf{H}$ returns a fresh random $C_1$ and a fresh random $r_0$ via $\mathsf{chain}_0$. This call to $\mathsf{chain}_0$ has not been done before because this would have invoked the $\mathsf{H}$ call in $G_0$, which is excluded by the hypothesis. Simulator's case 3: If $((C_j, v_j), (C, r_j), j) \in L$ for some $C_j, v_j, r_j$ and $0 \leq j < m$, then the current call to $\mathsf{H}(C, v_i)$ appends to a chain. Thus, a fresh random $C_{j+2} \| r_{j+1}$ is returned. The involved $\mathsf{chain}_{j+1}$ or $\mathsf{chain}_m$ has not been called before for the same reason as $\mathsf{chain}_0$ above. Simulator's case 4: A fresh random $C_{-1} \| r_{-2}$ is returned. To conclude, a fresh random value is returned in every case in $G_1$.

The previous proof shows that the games $G_0$ and $G_1$ are indistinguishable assuming the hypotheses H1, H2, and H3 hold. We will now bound the probability that they do not hold. Suppose that there are at most $q_{\mathsf{H}}$ direct queries to $\mathsf{H}$ and $q_{\mathsf{chain}_n}$ direct queries to $\mathsf{chain}_n$.

- When H1 does not hold, the distinguisher does an $\mathsf{H}$ call from a chain corresponding to an earlier or later $\mathsf{chain}_n$ call without having done the $\mathsf{H}$ calls starting from the beginning of the chain, by using the matching $C$ value. There are at most $\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n}\right)$ different $C$ values from $\mathsf{H}$, and the distinguisher has $q_{\mathsf{H}}$ attempts to hit a matching one, so the probability that H1 does not hold is at most $\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n}\right) \cdot q_{\mathsf{H}} / 2^{l'}$.
- The probability that H2 does not hold at the $q$-th call to $\mathsf{H}$ is at most the probability that a fresh random value in $\{0,1\}^{l'}$ collides with $q$ values in $\{0,1\}^{l'}$, hence $q/2^{l'}$. So in total, the probability that H2 does not hold is $\sum_{q=1}^{q_{\mathsf{H}}} q/2^{l'} = q_{\mathsf{H}}(q_{\mathsf{H}}+1)/2^{l'+1}$.
- The probability that H3 does not hold is at most the probability that among $q_{\mathsf{H}}$ random values in $\{0,1\}^{l'}$, two of them collide, so it is at most $q_{\mathsf{H}}(q_{\mathsf{H}}-1)/2^{l'+1}$.

Hence, the probability that $G_0$ and $G_1$ are distinguished is at most

$$\frac{\left(\sum_{n=0}^{m} n \cdot q_{\mathsf{chain}_n}\right) \cdot q_{\mathsf{H}} + q_{\mathsf{H}}^2}{2^{l'}}.$$

□

# C

# *Appendices to the Analysis of HPKE's Authenticated Mode*

## C.1 Single- and Two-User Definitions for AKEM

In this section, we give simplified variants of AKEM security notions which only consider a single user (insider security) or two users (outsider security). We show that these notions non-tightly imply their corresponding $n$-user notion given in Section 4.5.1.

As in Section 4.5.1, we distinguish between Outsider-CCA, Insider-CCA and Outsider-Auth. In all notions, we have a dedicated challenge oracle Chall, whereas AEncap and ADecap always reveal the real key (provided that ADecap is not queried on a challenge ciphertext).

We give games $(q_e, q_d, q_c)$-2-Outsider-CCA$_\ell$ and $(q_e, q_d, q_c)$-2-Outsider-CCA$_r$ in Listing C.1 and games $(q_e, q_d, q_c)$-2-Outsider-Auth$_\ell$ and $(q_e, q_d, q_c)$-2-Outsider-Auth$_r$ in Listing C.3. We also provide the $(q_e, q_d, q_c)$-1-Insider-CCA games in Listing C.2, where compared to the $(n, q_e, q_d, q_c)$-Insider-CCA games we omit the key generation oracle Gen since we need to generate only one challenge key and w.l.o.g. we can do this at the beginning of the game. In all games, adversary $\mathcal{A}$ is allowed to make $q_e$ queries to AEncap, $q_d$ queries to ADecap and $q_c$ queries to Chall. We define the advantage of $\mathcal{A}$ in each game as

$$\begin{aligned}
\mathsf{Adv}^{(q_e,q_d,q_c)\text{-2-Outsider-CCA}}_{\mathcal{A},\mathsf{AKEM}} &:= \big|\Pr[(q_e,q_d,q_c)\text{-2-Outsider-CCA}_\ell(\mathcal{A}) \Rightarrow 1] \\
&\quad - \Pr[(q_e,q_d,q_c)\text{-2-Outsider-CCA}_r(\mathcal{A}) \Rightarrow 1]\big| \,, \\
\mathsf{Adv}^{(q_e,q_d,q_c)\text{-1-Insider-CCA}}_{\mathcal{A},\mathsf{AKEM}} &:= \big|\Pr[(q_e,q_d,q_c)\text{-1-Insider-CCA}_\ell(\mathcal{A}) \Rightarrow 1] \\
&\quad - \Pr[(q_e,q_d,q_c)\text{-1-Insider-CCA}_r(\mathcal{A}) \Rightarrow 1]\big| \,, \\
\mathsf{Adv}^{(q_e,q_d,q_c)\text{-2-Outsider-Auth}}_{\mathcal{A},\mathsf{AKEM}} &:= \big|\Pr[(q_e,q_d,q_c)\text{-2-Outsider-Auth}_\ell(\mathcal{A}) \Rightarrow 1] \\
&\quad - \Pr[(q_e,q_d,q_c)\text{-2-Outsider-Auth}_r(\mathcal{A}) \Rightarrow 1]\big| \,.
\end{aligned}$$

For each security notion, we prove that the corresponding single- or two-user notion with one challenge query implies the $n$-user notion of Section 4.5.1 with multiple challenge queries. This is captured in Theorems C.1 to C.3. The proofs are given in Appendices C.1.1 to C.1.3.

Listing C.1: Games $(q_e, q_d, q_c)$-2-Outsider-CCA$_\ell$ and $(q_e, q_d, q_c)$-2-Outsider-CCA$_r$ for AKEM. Adversary $\mathcal{A}$ makes at most $q_e$ queries to AEncap, at most $q_d$ queries to ADecap and at most $q_c$ queries to Chall.

$(q_e, q_d, q_c)$-2-Outsider-CCA$_\ell$ and
$(q_e, q_d, q_c)$-2-Outsider-CCA$_r$

$(\mathrm{sk}_1, \mathrm{pk}_1) \xleftarrow{\$} \mathsf{Gen}$
$(\mathrm{sk}_2, \mathrm{pk}_2) \xleftarrow{\$} \mathsf{Gen}$
$\mathcal{C} \leftarrow \emptyset$
$b \xleftarrow{\$} \mathcal{A}^{\mathrm{AEncap},\mathrm{ADecap},\mathrm{Chall}}(\mathrm{pk}_1, \mathrm{pk}_2)$
**return** $b$

Oracle AEncap$(i \in [2], \mathrm{pk})$
$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk})$
**return** $(c, K)$

Oracle Chall$(i \in [2], j \in [2])$
$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk}_j)$
$K \xleftarrow{\$} \mathcal{K}$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{(\mathrm{pk}_i, \mathrm{pk}_j, c, K)\}$
**return** $(c, K)$

Oracle ADecap$(j \in [2], \mathrm{pk}, c)$
**if** $\exists K : (\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{C}$
  **return** $\perp$
$K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
**return** $K$

Listing C.2: Games $(q_e, q_d, q_c)$-1-Insider-CCA$_\ell$ and $(q_e, q_d, q_c)$-1-Insider-CCA$_r$ for AKEM. Adversary $\mathcal{A}$ makes at most $q_e$ queries to AEncap, at most $q_d$ queries to ADecap and at most $q_c$ queries to Chall.

$(q_e, q_d, q_c)$-1-Insider-CCA$_\ell$ and
$(q_e, q_d, q_c)$-1-Insider-CCA$_r$

$(\mathrm{sk}^*, \mathrm{pk}^*) \xleftarrow{\$} \mathsf{Gen}$
$\mathcal{C} \leftarrow \emptyset$
$b \xleftarrow{\$} \mathcal{A}^{\mathrm{AEncap},\mathrm{ADecap},\mathrm{Chall}}(\mathrm{pk}^*)$
**return** $b$

Oracle AEncap(pk)
$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}^*, \mathrm{pk})$
**return** $(c, K)$

Oracle Chall(sk)
$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}, \mathrm{pk}^*)$
$K \xleftarrow{\$} \mathcal{K}$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{(\mu(\mathrm{sk}), c, K)\}$
**return** $(c, K)$

Oracle ADecap$(\mathrm{pk}, c)$
**if** $\exists K : (\mathrm{pk}, c, K) \in \mathcal{C}$
  **return** $\perp$
$K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}^*, \mathrm{pk}, c)$
**return** $K$

Listing C.3: Games $(q_e, q_d, q_c)$-2-Outsider-Auth$_\ell$ and $(q_e, q_d, q_c)$-2-Outsider-Auth$_r$ for AKEM. Adversary $\mathcal{A}$ makes at most $q_e$ queries AEncap, at most $q_d$ queries ADecap and at most $q_c$ queries Chall.

$(q_e, q_d, q_c)$-2-Outsider-Auth$_\ell$ and
$(q_e, q_d, q_c)$-2-Outsider-Auth$_r$

$(\mathrm{sk}_1, \mathrm{pk}_1) \xleftarrow{\$} \mathsf{Gen}$
$(\mathrm{sk}_2, \mathrm{pk}_2) \xleftarrow{\$} \mathsf{Gen}$
$\mathcal{E} \leftarrow \emptyset$
$b \xleftarrow{\$} \mathcal{A}^{\mathrm{AEncap},\mathrm{ADecap},\mathrm{Chall}}(\mathrm{pk}_1, \mathrm{pk}_2)$
**return** $b$

Oracle AEncap$(i \in [2], \mathrm{pk})$
$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk})$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathrm{pk}_i, \mathrm{pk}, c, K)\}$
**return** $(c, K)$

Oracle ADecap$(j \in [2], \mathrm{pk}, c)$
**if** $\exists K : (\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{E}$
  **return** $\perp$
$K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathrm{pk}, \mathrm{pk}_j, c, K)\}$
**return** $K$

Oracle Chall$(i \in [2], j \in [2], c)$
**if** $\exists K : (\mathrm{pk}_i, \mathrm{pk}_j, c, K) \in \mathcal{E}$
  **return** $\perp$
$K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}_i, c)$
**if** $K \neq \perp$
  $K \xleftarrow{\$} \mathcal{K}$
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathrm{pk}_i, \mathrm{pk}_j, c, K)\}$
**return** $K$

**Theorem C.1** ($(q_e, q_d, 1)$-2-Outsider-CCA $\implies$ $(n, q_e, q_d)$-Outsider-CCA)**.** *For any adversary $\mathcal{A}$ against the $(n, q_e, q_d)$-Outsider-CCA security experiment, there exists an adversary $\mathcal{B}$ against the $(q_e, q_d, 1)$-2-Outsider-CCA security experiment such that $t_{\mathcal{B}} \approx t_{\mathcal{A}}$ and*

$$\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-CCA}}_{\mathcal{A},\mathsf{AKEM}} \leq n^2 q_e \cdot \mathsf{Adv}^{(q_e,q_d,1)\text{-2-Outsider-CCA}}_{\mathcal{B},\mathsf{AKEM}} + n^2 \cdot P_{\mathsf{AKEM}} .$$

**Theorem C.2** ($(q_e, q_d, 1)$-1-Insider-CCA $\implies$ $(n, q_e, q_d, q_c)$-Insider-CCA)**.** *For any adversary $\mathcal{A}$ against the $(n, q_e, q_d, q_c)$-Insider-CCA security experiment, there exists an adversary $\mathcal{B}$ against the $(q_e, q_d, 1)$-1-Insider-CCA security experiment such that $t_{\mathcal{B}} \approx t_{\mathcal{A}}$ and*

$$\mathsf{Adv}^{(n,q_e,q_d,q_c)\text{-Insider-CCA}}_{\mathcal{A},\mathsf{AKEM}} \leq n q_c \cdot \mathsf{Adv}^{(q_e,q_d,1)\text{-1-Insider-CCA}}_{\mathcal{B},\mathsf{AKEM}} .$$

**Theorem C.3** ($(q_e, q_d, 1)$-2-Outsider-Auth $\implies$ $(n, q_e, q_d)$-Outsider-Auth)**.** *For any adversary $\mathcal{A}$ against $(n, q_e, q_d)$-Outsider-Auth security of* AKEM*, there exists an adversary $\mathcal{B}$ against $(q_e, q_d, 1)$-2-Outsider-Auth security of* AKEM *such that $t_{\mathcal{B}} \approx t_{\mathcal{A}}$ and*

$$\mathsf{Adv}^{(n,q_e,q_d)\text{-Outsider-Auth}}_{\mathcal{A},\mathsf{AKEM}} \leq n^2 q_d \cdot \mathsf{Adv}^{(q_e,q_d,1)\text{-2-Outsider-Auth}}_{\mathcal{B},\mathsf{AKEM}} + n^2 \cdot P_{\mathsf{AKEM}} .$$

### C.1.1 *Proof of Theorem C.1*

*Proof.* Let $\mathcal{A}$ be an adversary against $(n, q_e, q_d)$-Outsider-CCA security of AKEM that makes at most $n$ queries to GEN, at most $q_e$ queries to AENCAP and $q_d$ queries to ADECAP. Consider the games $\mathsf{G}_0$-$\mathsf{G}_4$ in Listing C.4.

Listing C.4: Games $\mathsf{G}_0$-$\mathsf{G}_4$ for the proof of Theorem C.1.

$\mathsf{G}_0$, $\mathsf{G}_1$, $(\mathsf{G}_{2,u,v,q})_{u,v\in[n],q\in[q_c]_0}$, $\mathsf{G}_3$, $\mathsf{G}_4$

$ctr \leftarrow 0$ // $\mathsf{G}_{2,u,v,q}$
$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$\mathsf{COLL}_{\mathrm{pk}} \leftarrow$ **false**
$b \xleftarrow{\$} \mathcal{A}^{\mathrm{GEN},\mathrm{AENCAP},\mathrm{ADECAP}}$
**return** $b$

Oracle GEN
$\ell \leftarrow \ell + 1$
$(\mathrm{sk}_\ell, \mathrm{pk}_\ell) \xleftarrow{\$} \mathsf{Gen}$
**if** $\mathrm{pk}_\ell \in \{\mathrm{pk}_1, \ldots \mathrm{pk}_{\ell-1}\}$ // $\mathsf{G}_1$-$\mathsf{G}_3$
  $\mathsf{COLL}_{\mathrm{pk}} \leftarrow \mathrm{true}$; **abort** // $\mathsf{G}_1$-$\mathsf{G}_3$
**return** $(\mathrm{pk}_\ell, \ell)$

Oracle ADECAP$(j \in [\ell], \mathrm{pk}, c)$
**try get** $K$ s. t. $(\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{E}$
  **then return** $K$
$K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
**return** $K$

Oracle AENCAP$(i \in [\ell], \mathrm{pk})$
$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk})$
**if** $\mathrm{pk} \in \{\mathrm{pk}_1, \ldots, \mathrm{pk}_\ell\}$
  find $j$ such that $\mathrm{pk} = \mathrm{pk}_j$ // $\mathsf{G}_{2,u,v,q}$
  **if** $i < u$ **or** $(i = u \wedge j < v)$ // $\mathsf{G}_{2,u,v,q}$
    $K \xleftarrow{\$} \mathcal{K}$ // $\mathsf{G}_{2,u,v,q}$
  **if** $i = u$ **and** $j = v$ // $\mathsf{G}_{2,u,v,q}$
    $ctr \leftarrow ctr + 1$ // $\mathsf{G}_{2,u,v,q}$
    **if** $ctr \leq q$ // $\mathsf{G}_{2,u,v,q}$
      $K \xleftarrow{\$} \mathcal{K}$ // $\mathsf{G}_{2,u,v,q}$
  $K \xleftarrow{\$} \mathcal{K}$ // $\mathsf{G}_3$-$\mathsf{G}_4$
  $\mathcal{E} \leftarrow \mathcal{E} \uplus \{(\mathrm{pk}_i, \mathrm{pk}, c, K)\}$
**return** $(c, K)$

GAME $\mathsf{G}_0$. Note that this game is the $(n, q_e, q_d)$-Outsider-CCA$_\ell$ game. We already initialise multiset $\mathcal{E}$ and boolean flag $\mathsf{COLL}_{\mathrm{pk}}$, which we will need in the following games to exclude collisions. In $\mathsf{G}_0$, we model $\mathcal{E}$ as a multiset to store all challenge queries to AENCAP and to already match the syntax of the $(n, q_e, q_d)$-Outsider-CCA$_r$ game. If a challenge is queried to the decapsulation oracle ADECAP, then the challenge key stored in $\mathcal{E}$ is directly returned. Due to perfect correctness, this is only a conceptual change. We have

$$\Pr[\mathsf{G}_0 \Rightarrow 1] = \Pr[(n, q_e, q_d)\text{-Outsider-CCA}_\ell(\mathcal{A}) \Rightarrow 1] .$$

GAME $\mathsf{G}_1$. In this game, we raise flag $\mathsf{COLL}_{\mathrm{pk}}$ and abort, whenever a query to GEN generates a public key that was already generated in a previous query to GEN. Due to the difference lemma,

$$|\Pr[\mathsf{G}_0 \Rightarrow 1] - \Pr[\mathsf{G}_1 \Rightarrow 1]| \leq \Pr[\mathsf{COLL}_{\mathrm{pk}}] ,$$

which we can bound by

$$\Pr[\mathsf{COLL}_{\mathrm{pk}}] \leq \binom{n}{2} \cdot P_{\mathsf{AKEM}} \leq \frac{n^2}{2} \cdot P_{\mathsf{AKEM}} .$$

In the following we want to iterate over all pairs of honest users using indices $u, v \in [n]$ and all their challenges using index $q \in [q_e]_0$ and replace the corresponding challenge key by a random key. We capture each iteration in an intermediate game $\mathsf{G}_{2,u,v,q}$, where the key of the $q$-th query to AENCAP for sender public key $\mathrm{pk}_u$ and receiver public key $\mathrm{pk}_v$ is replaced by random.

At this point we want to remark that $q_e$ is the number of queries for *all* users and regardless whether it is a challenge query or not. In particular, the actual number of challenge queries for a specific user pair might be lower. However, in our following analysis we cannot easily use this fact to achieve a better bound because we have to cover all cases, which explains the tightness loss of $n^2 \cdot q_e$.

GAMES $(\mathsf{G}_{2,u,v,q})_{u,v\in[n],q\in[q_e]_0}$. As each public key generated by the game is unique, we can now find a unique index $j$ for queries to AENCAP such that the public key $\mathrm{pk}$ provided by $\mathcal{A}$ matches exactly one $\mathrm{pk}_j$. To identify the corresponding challenge query, we use a counter $ctr$. Each time, a challenge query for sender public key $\mathrm{pk}_u$ and receiver public key $\mathrm{pk}_v$ is issued, the counter is increased. Note that the first intermediate game $\mathsf{G}_{2,1,1,0}$ is exactly game $\mathsf{G}_1$, thus

$$\Pr[\mathsf{G}_{2,1,1,0} \Rightarrow 1] = \Pr[\mathsf{G}_1 \Rightarrow 1] .$$

Also note that whenever $q$ reaches $q_e$, there will be no change in the next step. In particular, for all $u \in [n]$, $v \in [n-1]$, $\mathsf{G}_{2,u,v,q_e}$ is the same as $\mathsf{G}_{2,u,v+1,0}$, as well as for all $u \in [n-1]$, $\mathsf{G}_{2,u,n,q_e}$ is the same as $\mathsf{G}_{2,u+1,1,0}$ and thus

$$\Pr[\mathsf{G}_{2,u,v,q_e} \Rightarrow 1] = \Pr[\mathsf{G}_{2,u,v+1,0} \Rightarrow 1] ,$$

$$\Pr[\mathsf{G}_{2,u,n,q_e} \Rightarrow 1] = \Pr[\mathsf{G}_{2,u+1,1,0} \Rightarrow 1] .$$

On a high level each game $\mathsf{G}_{2,u,v,q}$ separates the output of AENCAP on a query $(i, \mathrm{pk}_j)$, where $\mathrm{pk}_j \in \{\mathrm{pk}_1, \ldots, \mathrm{pk}_\ell\}$ is one of the public keys generated by the game so far, into two sets. The game outputs 1) a random key if $i < u$ or if $i = u$ and $j < v$ as well as if $i = u$ and $j = v$ and $ctr \leq q$, and 2) the real key otherwise.

We now need to bound the difference between games $\mathsf{G}_{2,u,v,q-1}$ and $\mathsf{G}_{2,u,v,q}$ for all $u, v \in [n]$, $q \in [q_e]$. Therefore, we construct adversary $\mathcal{B}_{u,v,q}$ against $(q_e, q_d, 1)$-2-Outsider-CCA in Listing C.5.

Adversary $\mathcal{B}_{u,v,q}$ has access to oracles `AEncap`, `ADecap` and `Chall`. It inputs two public keys which we denote by $\mathrm{pk}'_1$ and $\mathrm{pk}'_2$ in order to avoid confusion with public keys $\mathrm{pk}_1$ and $\mathrm{pk}_2$ which will be the first two public keys that $\mathcal{B}_{u,v,q}$ outputs to $\mathcal{A}$ when $\mathcal{A}$ queries GEN. On the $u$-th query to GEN, $\mathcal{B}_{u,v,q}$ will then output $\mathrm{pk}'_1$ and on the $v$-the query it will output $\mathrm{pk}'_2$. Note that when $u = v$, $\mathcal{B}_{u,v,q}$ will only use $\mathrm{pk}'_1$. This captures queries to AENCAP

Listing C.5: Adversary $\mathcal{B}_{u,v,q}$ against $(q_e, q_d, 1)$-2-Outsider-CCA, where $u, v \in [n]$, $q \in [q_e]$ and $f : \{u, v\} \mapsto \{1, 2\}$ is a function such that, when $u = v$, $f(u) = f(v) = 1$, and when $u \neq v$, $f(u) = 1$ and $f(v) = 2$.

$\mathcal{B}^{\texttt{AEncap},\texttt{ADecap},\texttt{Chall}}_{u,v,q}(\mathrm{pk}'_1, \mathrm{pk}'_2)$
01 $ctr \leftarrow 0$
02 $\ell \leftarrow 0$
03 $\mathcal{E} \leftarrow \emptyset$
04 $b \xleftarrow{\$} \mathcal{A}^{\textsc{Gen},\textsc{AEncap},\textsc{ADecap}}$
05 **return** $b$

Oracle $\textsc{Gen}$
06 $\ell \leftarrow \ell + 1$
07 **if** $\ell = u$
08 $\quad \mathrm{pk}_u \leftarrow \mathrm{pk}'_1$
09 **else if** $\ell = v$
10 $\quad \mathrm{pk}_v \leftarrow \mathrm{pk}'_2$
11 **else**
12 $\quad (\mathrm{sk}_\ell, \mathrm{pk}_\ell) \xleftarrow{\$} \mathsf{Gen}$
13 **if** $\mathrm{pk}_\ell \in \{\mathrm{pk}_1, \ldots \mathrm{pk}_{\ell-1}\}$
14 $\quad$ **abort**
15 **return** $(\mathrm{pk}_\ell, \ell)$

Oracle $\textsc{ADecap}(j \in [\ell], \mathrm{pk}, c)$
16 **try get** $K$ s. t. $(\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{E}$
17 $\quad$ **then return** $K$
18 **if** $j \in \{u, v\}$
19 $\quad K \leftarrow \texttt{ADecap}(f(j), \mathrm{pk}, c)$
20 **else**
21 $\quad K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
22 **return** $K$

Oracle $\textsc{AEncap}(i \in [\ell], \mathrm{pk})$
23 **if** $i \notin \{u, v\}$
24 $\quad (c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk})$
25 **else if** $i = u$ **and** $\mathrm{pk} = \mathrm{pk}_v$
26 $\quad ctr \leftarrow ctr + 1$
27 $\quad$ **if** $ctr = q$
28 $\quad\quad (c, K) \leftarrow \texttt{Chall}(f(i), f(v))$
29 $\quad$ **else**
30 $\quad\quad (c, K) \leftarrow \texttt{AEncap}(f(i), \mathrm{pk})$
31 **else**
32 $\quad (c, K) \leftarrow \texttt{AEncap}(f(i), \mathrm{pk})$
33 **if** $\mathrm{pk} \in \{\mathrm{pk}_1, \ldots, \mathrm{pk}_\ell\}$
34 $\quad$ find $j$ such that $\mathrm{pk} = \mathrm{pk}_j$
35 $\quad$ **if** $i < u$ **or** $(i = u \wedge j < v)$
$\quad$ **or** $(i = u \wedge j = v \wedge ctr < q)$
36 $\quad\quad K \xleftarrow{\$} \mathcal{K}$
37 $\quad \mathcal{E} \leftarrow \mathcal{E} \uplus \{(\mathrm{pk}_i, \mathrm{pk}, c, K)\}$
38 **return** $(c, K)$

of the form $(i, \mathrm{pk}_i)$. All remaining key pairs will be honestly generated with the condition that $\mathcal{B}_{u,v,q}$ aborts whenever the same key is generated twice, as introduced in game $\mathsf{G}_1$. In order to avoid convoluted case distinctions in Listing C.5, we define a function $f : \{u, v\} \mapsto \{1, 2\}$ such that, when $u = v$, $f(u) = f(v) = 1$, and when $u \neq v$, $f(u) = 1$ and $f(v) = 2$. This ensures that oracles `AEncap`, `ADecap` and `Chall` are queried on the correct indices.

Queries to $\textsc{ADecap}$ are simulated using the `ADecap` oracle whenever $\mathcal{A}$ makes a query on $u$ or $v$. Otherwise, $\mathcal{B}_{u,v,q}$ can run the AuthDecap algorithm, since it knows the secret key $\mathrm{sk}_j$.

Queries to $\textsc{AEncap}$ are simulated as follows: We first want to run the AuthEncap algorithm for all queries, except for the $q$-th challenge query of sender $\mathrm{pk}_u$ and receiver $\mathrm{pk}_v$. Thus, if the index $i$ supplied by $\mathcal{A}$ is not $u$ or $v$, $\mathcal{B}_{u,v,q}$ can run the AuthEncap algorithm itself, since it knows the secret key $\mathrm{sk}_i$. For all other queries, $\mathcal{B}_{u,v,q}$ needs to calls its `AEncap` oracle. However, if this is the $q$-th challenge query of sender $\mathrm{pk}_u$ and receiver $\mathrm{pk}_v$, $\mathcal{B}_{u,v,q}$ calls the challenge oracle `Chall` instead, after incrementing $ctr$. At this point note that $\mathcal{B}_{u,v,q}$ makes at most $q_e$ queries to the `AEncap` oracle and only a single query to `Chall`.

We check for public key collisions and $\mathcal{B}_{u,v,q}$ aborts whenever $\mathsf{G}_{2,u,v,q}$ would abort. Now we can proceed to replacing all challenge keys which were considered in previous games. In particular, if $i < u$ or $i = u$ and $j < v$, where $\mathrm{pk} = \mathrm{pk}_j$ for some index $j \in [\ell]$, we choose a random key. We do the same in case $i = u$ and $j = v$ and $ctr < q$. Note that if $\mathcal{B}_{u,v,q}$ is in the $(q_e, q_d, 1)$-2-Outsider-CCA$_\ell$ game, it perfectly simulates $\mathsf{G}_{2,u,v,q-1}$. Otherwise, if $\mathcal{B}_{u,v,q}$ is in the $(q_e, q_d, 1)$-2-Outsider-CCA$_r$ game, it perfectly

simulates $\mathsf{G}_{2,u,v,q}$. Thus for all $q \in [q_e]$,

$$\left|\Pr[\mathsf{G}_{2,u,v,q-1} \Rightarrow 1] - \Pr[\mathsf{G}_{2,u,v,q} \Rightarrow 1]\right| \leq \mathsf{Adv}^{(q_e,q_d,1)\text{-2-Outsider-CCA}}_{\mathcal{B}_{u,v,q},\mathsf{AKEM}} .$$

GAME $\mathsf{G}_3$. In this game, all challenge keys output by AENCAP are random keys. Note that this game is the same as $\mathsf{G}_{2,u,v,q_e}$, hence

$$\Pr[\mathsf{G}_3 \Rightarrow 1] = \Pr[\mathsf{G}_{2,n,n,q_e} \Rightarrow 1] .$$

GAME $\mathsf{G}_4$. In game $\mathsf{G}_4$ we undo the change introduced in game $\mathsf{G}_1$ and do not abort if a public key collision happens. Thus, we get

$$\left|\Pr[\mathsf{G}_3 \Rightarrow 1] - \Pr[\mathsf{G}_4 \Rightarrow 1]\right| \leq \frac{n^2}{2} \cdot P_{\mathsf{AKEM}} .$$

Finally note that $\mathsf{G}_4$ is exactly the same as the $(n, q_e, q_d)$-Outsider-CCA$_r$ game and

$$\Pr[\mathsf{G}_4 \Rightarrow 1] = \Pr[(n, q_e, q_d)\text{-Outsider-CCA}_r(\mathcal{A}) \Rightarrow 1] .$$

Folding adversaries $\mathcal{B}_{u,v,q}$ into a single adversary $\mathcal{B}$ and collecting the probabilities yields the bound claimed in Theorem C.1. □

### C.1.2 *Proof of Theorem C.2*

*Proof.* Let $\mathcal{A}$ be an adversary in the $(n, q_e, q_d, q_c)$-Insider-CCA security experiment that makes at most $n$ queries to GEN, at most $q_e$ queries to AENCAP, $q_d$ queries to ADECAP and $q_c$ queries to CHALL. Consider the sequence of games $\mathsf{G}_0$-$\mathsf{G}_2$ in Listing C.6.

Listing C.6: Games $\mathsf{G}_0$-$\mathsf{G}_2$ for the proof of Theorem C.2.

$\underline{\mathsf{G}_0, (\mathsf{G}_{1,u,q})_{u\in[n],q\in[q_c]_0}, \mathsf{G}_2}$
01 $ctr \leftarrow 0$
02 $\ell \leftarrow 0$
03 $\mathcal{C} \leftarrow \emptyset$
04 $b \xleftarrow{\$} \mathcal{A}^{\text{GEN},\text{AENCAP},\text{ADECAP},\text{CHALL}}$
05 **return** $b$

$\underline{\text{Oracle GEN}}$
06 $\ell \leftarrow \ell + 1$
07 $(\mathrm{sk}_\ell, \mathrm{pk}_\ell) \xleftarrow{\$} \mathsf{Gen}$
08 **return** $(\mathrm{pk}_\ell, \ell)$

$\underline{\text{Oracle ADECAP}(j \in [\ell], \mathrm{pk}, c)}$
09 **try get** $K$ s. t. $(\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{C}$
10 **then return** $K$
11 $K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
12 **return** $K$

$\underline{\text{Oracle AENCAP}(i \in [\ell], \mathrm{pk})}$
13 $(c, K) \leftarrow \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk})$
14 **return** $(c, K)$

$\underline{\text{Oracle CHALL}(j \in [\ell], \mathrm{sk})}$
15 $(c, K) \leftarrow \mathsf{AuthEncap}(\mathrm{sk}, \mathrm{pk}_j)$
16 **if** $j < u$ // $\mathsf{G}_{1,u,q}$
17 $K \xleftarrow{\$} \mathcal{K}$ // $\mathsf{G}_{1,u,q}$
18 **if** $j = u$ // $\mathsf{G}_{1,u,q}$
19 $ctr \leftarrow ctr + 1$ // $\mathsf{G}_{1,u,q}$
20 **if** $ctr \leq q$ // $\mathsf{G}_{1,u,q}$
21 $K \xleftarrow{\$} \mathcal{K}$ // $\mathsf{G}_{1,u,q}$
22 $K \xleftarrow{\$} \mathcal{K}$ // $\mathsf{G}_2$
23 $\mathcal{C} \leftarrow \mathcal{C} \uplus \{(\mu(\mathrm{sk}), \mathrm{pk}_j, c, K)\}$
24 **return** $(c, K)$

GAME $\mathsf{G}_0$. This is the original $(n, q_e, q_d, q_c)$-Insider-CCA$_\ell$ game. The set $\mathcal{C}$ is used to store all queries to the challenge oracle CHALL. If a challenge is queried to the decapsulation oracle ADECAP, then the challenge key is directly returned. Due to perfect correctness, this is only a conceptual change. Thus,

$$\Pr[\mathsf{G}_0 \Rightarrow 1] = \Pr[(n, q_e, q_d, q_c)\text{-Insider-CCA}_\ell(\mathcal{A}) \Rightarrow 1] .$$

In the following, we want to replace all real challenge keys by random keys, iterating over all users (indicated by index $u$) and their challenges (indicated by index $q$). We capture each iteration in an intermediate game $\mathsf{G}_{1,u,q}$, where the key of the $q$-th query to CHALL for receiver public key $\mathrm{pk}_u$ is replaced by random.

At this point we want to remark that $q_c$ is the total number of challenge queries for *all* users. In particular, the actual number of challenge queries for a specific user might be lower. However, in our following analysis we cannot easily use this fact to achieve a better bound because we have to cover all cases, which explains the tightness loss of $n \cdot q_c$.

GAMES $(\mathsf{G}_{1,u,q})_{u\in[n],q\in[q_c]_0}$. To identify the corresponding challenge query, we use a counter $ctr$. Each time, a challenge query for receiver public key $\mathrm{pk}_u$ is issued, the counter is increased. Note that the first intermediate game $\mathsf{G}_{1,1,0}$ is the same as $\mathsf{G}_0$ as well as $\mathsf{G}_{1,u,q_c}$ is the same as $\mathsf{G}_{1,u+1,0}$ for all $u \in [n-1]$. Hence,

$$\Pr[\mathsf{G}_{1,1,0} \Rightarrow 1] = \Pr[\mathsf{G}_0 \Rightarrow 1] \,,$$

and

$$\Pr[\mathsf{G}_{1,u,q_c} \Rightarrow 1] = \Pr[\mathsf{G}_{1,u+1,0} \Rightarrow 1] \,.$$

In order to bound the difference between $\mathsf{G}_{1,u,q-1}$ and $\mathsf{G}_{1,u,q}$ for $u \in [n]$, $q \in [q_c]$, we construct an adversary $\mathcal{B}_{u,q}$ against $(q_e, q_d, 1)$-1-Insider-CCA in Listing C.7.

Listing C.7: Adversary $\mathcal{B}_{u,q}$ against $(q_e, q_d, 1)$-1-Insider-CCA, where $u \in [n]$ and $q \in [q_c]$.

$\mathcal{B}^{\mathtt{AEncap},\mathtt{ADecap},\mathtt{Chall}}_{u,q}(\mathrm{pk}^*)$
01 $ctr \leftarrow 0$
02 $\ell \leftarrow 0$
03 $\mathcal{C} \leftarrow \emptyset$
04 $b \xleftarrow{\$} \mathcal{A}^{\mathrm{GEN},\mathrm{AENCAP},\mathrm{ADECAP},\mathrm{CHALL}}$
05 **return** $b$

Oracle GEN
06 $\ell \leftarrow \ell + 1$
07 **if** $\ell = u$
08 $\quad \mathrm{pk}_u \leftarrow \mathrm{pk}^*$
09 **else**
10 $\quad (\mathrm{sk}_\ell, \mathrm{pk}_\ell) \xleftarrow{\$} \mathsf{Gen}$
11 **return** $(\mathrm{pk}_\ell, \ell)$

Oracle ADECAP$(j \in [\ell], \mathrm{pk}, c)$
12 **try get** $K$ s. t. $(\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{C}$
13 $\quad$ **then return** $K$
14 **if** $j = u$
15 $\quad K \leftarrow \mathtt{ADecap}(\mathrm{pk}, c)$
16 **else**
17 $\quad K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
18 **return** $K$

Oracle AENCAP$(i \in [\ell], \mathrm{pk})$
19 **if** $i = u$
20 $\quad (c, K) \leftarrow \mathtt{AEncap}(\mathrm{pk})$
21 **else**
22 $\quad (c, K) \leftarrow \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk})$
23 **return** $(c, K)$

Oracle CHALL$(j \in [\ell], \mathrm{sk})$
24 $(c, K) \leftarrow \mathsf{AuthEncap}(\mathrm{sk}, \mathrm{pk}_j)$
25 **if** $j = u$
26 $\quad ctr \leftarrow ctr + 1$
27 $\quad$ **if** $ctr = q$
28 $\quad\quad (c, K) \leftarrow \mathtt{Chall}(\mathrm{sk})$
29 **if** $j < u$ **or** $(j = u \wedge ctr < q)$
30 $\quad K \xleftarrow{\$} \mathcal{K}$
31 $\mathcal{C} \leftarrow \mathcal{C} \uplus \{(\mu(\mathrm{sk}), \mathrm{pk}_j, c, K)\}$
32 **return** $(c, K)$

$\mathcal{B}_{u,q}$ inputs a challenge public key $\mathrm{pk}^*$ and has access to oracles `AEncap`, `ADecap`, `Chall`. It initialises counters $ctr$ for the number of challenge queries and $\ell$ for the number of users. On the $u$-th query to GEN, $\mathcal{B}_{u,q}$ outputs $\mathrm{pk}^*$ to $\mathcal{A}$. On all other queries, it samples a new key pair. If $\mathcal{A}$ queries AENCAP on index $u$, $\mathcal{B}_{u,q}$ uses its `AEncap` oracle to answer the query. For all other indices, it knows the corresponding secret key and can run the AuthEncap algorithm on its own. The same applies for queries to the ADECAP oracle.

If $\mathcal{A}$ issues a challenge query on an index $j$ and a secret key $\mathrm{sk}$, $\mathcal{B}_{u,q}$ first runs the AuthEncap algorithm itself. If $j = u$, the counter $ctr$ is incremented and if this is the $q$-th query, $\mathcal{B}_{u,q}$ queries the challenge oracle `Chall`. It then checks whether the challenge key needs to be replaced by a random key, that means in case $j < u$ or in case $j = u$ and $ctr < q$.

If $\mathcal{B}_{u,q}$ is in the $(q_e, q_d, 1)$-1-Insider-CCA$_\ell$ game, it perfectly simulates $\mathsf{G}_{1,u,q-1}$. Otherwise, if $\mathcal{B}_{u,q}$ is in the $(q_e, q_d, 1)$-1-Insider-CCA$_r$ game, it perfectly simulates $\mathsf{G}_{1,u,q}$, hence

$$\left|\Pr[\mathsf{G}_{1,u,q-1} \Rightarrow 1] - \Pr[\mathsf{G}_{1,u,q} \Rightarrow 1]\right| \leq \mathsf{Adv}^{(q_e,q_d,1)\text{-1-Insider-CCA}}_{\mathcal{B}_{u,q},\mathsf{AKEM}} .$$

GAME $\mathsf{G}_2$. In this game, all keys output by the challenge oracle CHALL are random keys. Note that this game is the same as $\mathsf{G}_{1,n,q_c}$, hence

$$\Pr[\mathsf{G}_2 \Rightarrow 1] = \Pr[\mathsf{G}_{1,n,q_c} \Rightarrow 1] .$$

Finally note that $\mathsf{G}_2$ is exactly the same as the $(n, q_e, q_d, q_c)$-Insider-CCA$_r$ game and

$$\Pr[\mathsf{G}_2 \Rightarrow 1] = \Pr[(n, q_e, q_d, q_c)\text{-Insider-CCA}_r(\mathcal{A}) \Rightarrow 1] .$$

Folding adversaries $\mathcal{B}_{u,q}$ into a single adversary $\mathcal{B}$ and collecting the probabilities yields the bound claimed in Theorem C.2. □

### C.1.3 *Proof of Theorem C.3*

*Proof.* Let $\mathcal{A}$ be an adversary against $(n, q_e, q_d)$-Outsider-Auth security of AKEM that makes at most $n$ queries to GEN, at most $q_e$ queries to AENCAP and $q_d$ queries to ADECAP. Consider the games $\mathsf{G}_0$-$\mathsf{G}_4$ in Listing C.8. We proceed similarly to the proof of Theorem C.1.

Listing C.8: Games $\mathsf{G}_0$-$\mathsf{G}_4$ for the proof of Theorem C.3, where $u \in [n], v \in [n]_0$.

```
G_0, G_1, (G_{2,u,v})_{u∈[n],v∈[n]_0}, G_3, G_4
01 ctr ← 0                                          // G_{2,u,v,q}
02 ℓ ← 0
03 E ← ∅
04 COLL_pk ← false
05 b ←$ A^{GEN,AENCAP,ADECAP}
06 return b

Oracle GEN
07 ℓ ← ℓ + 1
08 (sk_ℓ, pk_ℓ) ←$ Gen
09 if pk_ℓ ∈ {pk_1, ... pk_{ℓ−1}}                   // G_1-G_3
10    COLL_pk ← true; abort                         // G_1-G_3
11 return (pk_ℓ, ℓ)

Oracle AENCAP(i ∈ [ℓ], pk)
12 (c, K) ←$ AuthEncap(sk_i, pk)
13 E ← E ⊎ {(pk_i, pk, c, K)}
14 return (c, K)

Oracle ADECAP(j ∈ [ℓ], pk, c)
15 try get K s.t. (pk, pk_j, c, K) ∈ E
16    then return K
17 K ← AuthDecap(sk_j, pk, c)
18 if pk ∈ {pk_1, ..., pk_ℓ} and K ≠ ⊥
19    find i such that pk = pk_i                    // G_{2,u,v,q}
20    if j < u or (j = u ∧ i < v)                   // G_{2,u,v,q}
21       K ←$ K                                     // G_{2,u,v,q}
22    if j = u ∧ i = v                              // G_{2,u,v,q}
23       ctr ← ctr + 1                              // G_{2,u,v,q}
24       if ctr ≤ q                                 // G_{2,u,v,q}
25          K ←$ K                                  // G_{2,u,v,q}
26    K ←$ K                                        // G_3-G_4
27    E ← E ⊎ {(pk_i, pk_j, c, K)}
28 return K
```

GAME $\mathsf{G}_0$. Note that this game is the $(n, q_e, q_d)$-Outsider-Auth$_\ell$ game. We already initialise the multiset $\mathcal{E}$ and boolean flag $\mathsf{COLL}_{\mathrm{pk}}$, which we will need in the following games. In $\mathsf{G}_0$, $\mathcal{E}$ is used to store all queries to the encapsulation oracle AENCAP and all queries to the decapsulation oracle

ADECAP, which correspond to challenge queries, that is all queries where $\mathrm{pk}$ supplied by $\mathcal{A}$ is one of the public keys generated by the game. If a challenge or a previous AENCAP query is queried to ADECAP again, then the corresponding key is directly returned. Due to perfect correctness, this is only a conceptual change. Thus,

$$\Pr[\mathsf{G}_0 \Rightarrow 1] = \Pr[(n, q_e, q_d)\text{-Outsider-Auth}_\ell(\mathcal{A}) \Rightarrow 1] \,.$$

GAME $\mathsf{G}_1$. In this game, we raise flag $\mathsf{COLL}_{\mathrm{pk}}$ and abort, whenever a query to GEN generates a public key that was already generated in a previous query to GEN. Due to the difference lemma,

$$|\Pr[\mathsf{G}_0 \Rightarrow 1] - \Pr[\mathsf{G}_1 \Rightarrow 1]| \leq \Pr[\mathsf{COLL}_{\mathrm{pk}}] \,,$$

which we can bound by

$$\Pr[\mathsf{COLL}_{\mathrm{pk}}] \leq \binom{n}{2} \cdot P_{\mathsf{AKEM}} \leq \frac{n^2}{2} \cdot P_{\mathsf{AKEM}} \,.$$

In the following we want to iterate over all pairs of honest users using indices $u, v \in [n]$ and all their challenges using index $q \in [q_d]_0$ and replace the corresponding challenge key by a random key. We capture each iteration in an intermediate game $\mathsf{G}_{2,u,v,q}$, where the key of the $q$-th query to ADECAP for sender public key $\mathrm{pk}_u$ and receiver public key $\mathrm{pk}_v$ is replaced by random.

At this point we want to remark that $q_d$ is the number of queries for *all* users and regardless whether it is a challenge query or not. In particular, the actual number of challenge queries for a specific user pair might be lower. However, in our following analysis we cannot easily use this fact to achieve a better bound because we have to cover all cases, which explains the tightness loss of $n^2 \cdot q_d$.

GAMES $(\mathsf{G}_{2,u,v,q})_{u,v\in[n],q\in[q_d]_0}$. To identify the corresponding challenge query, we use a counter $ctr$. Each time a challenge query for receiver public key $\mathrm{pk}_u$ and sender public key $\mathrm{pk}_v$ is issued, the counter is increased. Note that the first intermediate game $\mathsf{G}_{2,1,1,0}$ is exactly game $\mathsf{G}_1$, thus

$$\Pr[\mathsf{G}_{2,1,1,0} \Rightarrow 1] = \Pr[\mathsf{G}_1 \Rightarrow 1] \,.$$

Also note that whenever $q$ reaches $q_d$, there will be no change in the next step. In particular, for all $u \in [n]$, $v \in [n-1]$, $\mathsf{G}_{2,u,v,q_d}$ is the same as $\mathsf{G}_{2,u,v+1,0}$, as well as for all $u \in [n-1]$, $\mathsf{G}_{2,u,n,q_e}$ is the same as $\mathsf{G}_{2,u+1,1,0}$ and thus

$$\Pr[\mathsf{G}_{2,u,v,q_d} \Rightarrow 1] = \Pr[\mathsf{G}_{2,u,v+1,0} \Rightarrow 1] \,,$$

$$\Pr[\mathsf{G}_{2,u,n,q_d} \Rightarrow 1] = \Pr[\mathsf{G}_{2,u+1,1,0} \Rightarrow 1] \,.$$

On a high level each game $\mathsf{G}_{2,u,v,q}$ separates the output of ADECAP on a query $(j, \mathrm{pk}_i)$, where $\mathrm{pk}_i \in \{\mathrm{pk}_1, \ldots, \mathrm{pk}_\ell\}$ is one of the public keys generated by the game so far, into two sets. The game outputs 1) a random key if $j < u$ or if $j = u$ and $i < v$ as well as if $j = u$ and $i = v$ and $ctr \leq q$, and 2) the real key otherwise.

At this point note that for challenges the index $i$ is uniquely defined as each public key generated by the game is unique. Also, each entry in the multiset $\mathcal{E}$ is unique.

Listing C.9: Adversary $\mathcal{B}_{u,v,q}$ against $(q_e, q_d, 1)$-2-Outsider-Auth, where $f : \{u, v\} \mapsto \{1, 2\}$ is a function such that, when $u = v$, $f(u) = f(v) = 1$, and when $u \neq v$, $f(u) = 1$ and $f(v) = 2$.

$\mathcal{B}^{\texttt{AEncap},\texttt{ADecap},\texttt{Chall}}_{u,v,q}(\mathrm{pk}'_1, \mathrm{pk}'_2)$
$ctr \leftarrow 0$
$\ell \leftarrow 0$
$\mathcal{E} \leftarrow \emptyset$
$b \xleftarrow{\$} \mathcal{A}^{\text{GEN},\text{AENCAP},\text{ADECAP}}$
**return** $b$

Oracle GEN
$\ell \leftarrow \ell + 1$
**if** $\ell = u$
  $\mathrm{pk}_u \leftarrow \mathrm{pk}'_1$
**else if** $\ell = v$
  $\mathrm{pk}_v \leftarrow \mathrm{pk}'_2$
**else**
  $(\mathrm{sk}_\ell, \mathrm{pk}_\ell) \xleftarrow{\$} \mathsf{Gen}$
**if** $\mathrm{pk}_\ell \in \{\mathrm{pk}_1, \ldots \mathrm{pk}_{\ell-1}\}$
  **abort**
**return** $(\mathrm{pk}_\ell, \ell)$

Oracle AENCAP$(i \in [\ell], \mathrm{pk})$
**if** $i \in \{u, v\}$
  $(c, K) \leftarrow \texttt{AEncap}(f(i), \mathrm{pk})$
**else**
  $(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(\mathrm{sk}_i, \mathrm{pk})$
$\mathcal{E} \leftarrow \mathcal{E} \uplus \{(\mathrm{pk}_i, \mathrm{pk}, c, K)\}$
**return** $(c, K)$

Oracle ADECAP$(j \in [\ell], \mathrm{pk}, c)$
**try get** $K$ s. t. $(\mathrm{pk}, \mathrm{pk}_j, c, K) \in \mathcal{E}$
  **then return** $K$
**if** $\mathrm{pk} \notin \{\mathrm{pk}_1, \ldots, \mathrm{pk}_\ell\}$
  **if** $j \in \{u, v\}$
    $K \leftarrow \texttt{ADecap}(f(j), \mathrm{pk}, c)$
  **else**
    $K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
  **return** $K$
find $i$ such that $\mathrm{pk} = \mathrm{pk}_i$
**if** $j \notin \{u, v\}$
  $K \leftarrow \mathsf{AuthDecap}(\mathrm{sk}_j, \mathrm{pk}, c)$
**else if** $j = u$ **and** $i = v$
  $ctr \leftarrow ctr + 1$
  **if** $ctr = q$
    $K \leftarrow \texttt{Chall}(f(j), f(i), c)$
  **else**
    $K \leftarrow \texttt{ADecap}(f(j), \mathrm{pk}_i, c)$
**else**
  $K \leftarrow \texttt{ADecap}(f(j), \mathrm{pk}_i, c)$
**if** $j < u$ **or** $(j = u \wedge i < v)$
  **or** $(j = u \wedge i = v \wedge ctr < q)$
  **if** $K \neq \bot$
    $K \xleftarrow{\$} \mathcal{K}$
**if** $K \neq \bot$
  $\mathcal{E} \leftarrow \mathcal{E} \uplus \{(\mathrm{pk}, \mathrm{pk}_j, c, K)\}$
**return** $K$

We now need to bound the difference between games $\mathsf{G}_{2,u,v,q-1}$ and $\mathsf{G}_{2,u,v,q}$ for all $u, v \in [n]$, $q \in [q_d]$. Therefore, we construct adversary $\mathcal{B}_{u,v,q}$ against $(q_e, q_d, 1)$-2-Outsider-Auth in Listing C.9.

Adversary $\mathcal{B}_{u,v,q}$ has access to oracles `AEncap`, `ADecap` and `Chall`. It inputs two public keys which we denote by $\mathrm{pk}'_1$ and $\mathrm{pk}'_2$ in order to avoid confusion with public keys $\mathrm{pk}_1$ and $\mathrm{pk}_2$ which will be the first two public keys that $\mathcal{B}_{u,v,q}$ outputs to $\mathcal{A}$ when $\mathcal{A}$ queries GEN. On the $u$-th query to GEN, $\mathcal{B}_{u,v,q}$ will then output $\mathrm{pk}'_1$ and on the $v$-the query it will output $\mathrm{pk}'_2$. Note that when $u = v$, $\mathcal{B}_{u,v,q}$ will only use $\mathrm{pk}'_1$. This captures queries to ADECAP of the form $(j, \mathrm{pk}_j)$. All remaining key pairs will be honestly generated with the condition that $\mathcal{B}_{u,v,q}$ aborts whenever the same key is generated twice, as introduced in game $\mathsf{G}_1$. As in the proof of Theorem C.1, we define a function $f : \{u, v\} \mapsto \{1, 2\}$ such that, when $u = v$, $f(u) = f(v) = 1$, and when $u \neq v$, $f(u) = 1$ and $f(v) = 2$. This ensures that oracles `AEncap`, `ADecap` and `Chall` are queried on the correct indices.

Queries to AENCAP are simulated using the `AEncap` oracle whenever $\mathcal{A}$ makes a query on $u$ or $v$. Otherwise, $\mathcal{B}_{u,v,q}$ can run the $\mathsf{AuthEncap}$ algorithm, since it knows the secret key $\mathrm{sk}_i$.

Queries to ADECAP are simulated as follows: We first take care of non-challenges, that means where the public key supplied by $\mathcal{A}$ is not one of the public keys generated by $\mathcal{B}_{u,v,q}$ so far. If the index $j$ supplied by $\mathcal{A}$ is either $u$ or $v$, $\mathcal{B}_{u,v,q}$ queries its `ADecap` oracle. Otherwise, it knows the corresponding secret key and can run the $\mathsf{AuthDecap}$ algorithm itself. The result will then returned to $\mathcal{A}$. If the query was a challenge query, then $\mathrm{pk}$ must be $\mathrm{pk}_i$ for some unique $i \in [\ell]$. Before we can replace any challenge keys, we first need to compute the output of $\mathsf{AuthDecap}$ using the algorithm itself or the correct

oracle. If $j \notin \{u, v\}$, $\mathcal{B}_{u,v,q}$ knows the secret key and can thus compute the output on its own. If $j = u$ and $i = v$, we need to increment the counter $ctr$ and if this is the $q$-th query, $\mathcal{B}_{u,v,q}$ calls the challenge oracle `Chall`. In all other cases, it calls `ADecap`. At this point note that $\mathcal{B}_{u,v,q}$ makes at most $q_d$ queries to the `ADecap` oracle and only a single query to `Chall`.

Now we can proceed to replacing all challenge keys which were considered in previous games. In particular, if $j < u$ or $j = u$ and $i < v$, we choose a random key. We do the same in case $j = u$ and $i = v$ and $ctr < q$. Note that if $\mathcal{B}_{u,v,q}$ is in the $(q_e, q_d, 1)$-2-Outsider-Auth$_\ell$ game, it perfectly simulates $\mathsf{G}_{2,u,v,q-1}$. Otherwise, if $\mathcal{B}_{u,v,q}$ is in the $(q_e, q_d, 1)$-2-Outsider-Auth$_r$ game, it perfectly simulates $\mathsf{G}_{2,u,v,q}$. Thus for all $q \in [q_d]$,

$$\left|\Pr[\mathsf{G}_{2,u,v,q-1} \Rightarrow 1] - \Pr[\mathsf{G}_{2,u,v,q} \Rightarrow 1]\right| \leq \mathsf{Adv}^{(q_e,q_d,1)\text{-2-Outsider-Auth}}_{\mathcal{B}_{u,v,q},\mathsf{AKEM}} .$$

GAME $\mathsf{G}_3$. In this game, all challenge keys output by the oracle ADECAP are random keys. Note that this game is the same as $\mathsf{G}_{2,n,n,q_d}$, hence

$$\Pr[\mathsf{G}_3 \Rightarrow 1] = \Pr[\mathsf{G}_{2,n,n,q_d} \Rightarrow 1] .$$

GAME $\mathsf{G}_4$. In game $\mathsf{G}_4$ we undo the change introduced in game $\mathsf{G}_1$ and do not abort if a public key collision happens. Thus, we get

$$\left|\Pr[\mathsf{G}_3 \Rightarrow 1] - \Pr[\mathsf{G}_4 \Rightarrow 1]\right| \leq \frac{n^2}{2} \cdot P_{\mathsf{AKEM}} .$$

Finally note that $\mathsf{G}_4$ is exactly the same as the $(n, q_e, q_d)$-Outsider-Auth$_r$ game. We get

$$\Pr[\mathsf{G}_4 \Rightarrow 1] = \Pr[(n, q_e, q_d)\text{-Outsider-Auth}_r(\mathcal{A}) \Rightarrow 1] .$$

Folding adversaries $\mathcal{B}_{u,v,q}$ into a single adversary $\mathcal{B}$ and collecting the probabilities yields the bound claimed in Theorem C.3. □

## C.2 COMPARISON OF PSEUDOCODE DEFINITIONS AND THEIR IMPLEMENTATION IN CRYPTOVERIF

The goal of this section is to convince the readers that the CryptoVerif models provided as supplementary material actually prove the theorems presented in the main body of Chapter 4. Rather than explaining syntax and semantics of CryptoVerif on an abstract level, we take one of the chapter's security notions and explain by example how the pseudocode presented in the main body relates to the code written in CryptoVerif. The security notion we use as example is $(n, q_e, q_d)$-Outsider-CCA of AKEM, presented in Listing 4.3. This notion appears in Theorem 4.7, where we show that DH-AKEM satisfies it, and it appears in the composition theorems 4.3 and 4.5, where we use it as an assumption. In Listing C.10, we display the pseudocode definition of $(n, q_e, q_d)$-Outsider-CCA from Listing 4.3 in the left column next to its implementation as an assumption in CryptoVerif in the right column. The modelling of assumptions and proof goals in CryptoVerif has only some slight syntactical differences. Both columns contain line numbers, and we will use `Lxy` to refer to line number `xy` in the left column, and `Rxy` to refer to line number `xy` in the right column. As with all the listings throughout the paper

so far, the game of the real version of the notion contains only the non-boxed lines, and the game of the ideal version additionally contains the boxed lines. The complete rolled-out version of CryptoVerif code for the $(n, q_e, q_d)$-Outsider-CCA notion as assumption can be found in lib.authkem.ocvl, from lines 102 to 143. The CryptoVerif files are available at [Alw+].

[Alw+] Alwen et al., *Analysing the HPKE Standard Supplementary Material*

Listing C.10: Games $(n, q_e, q_d)$-Outsider-CCA$_\ell$ and $(n, q_e, q_d)$-Outsider-CCA$_r$ for AKEM, with their modelling in CryptoVerif shown on the right side.

$(n, q_e, q_d)$-Outsider-CCA$_\ell$ and
$(n, q_e, q_d)$-Outsider-CCA$_r$

```
01 ℓ ← 0
02 E ← ∅
03 b ←$ A^{Gen,AEncap,ADecap}
04 return b

Oracle Gen
05 ℓ ← ℓ + 1
06 (sk_ℓ, pk_ℓ) ←$ Gen
07 return (pk_ℓ, ℓ)

Oracle AEncap(i ∈ [ℓ], pk)
08 (c, K) ←$ AuthEncap(sk_i, pk)
09 if pk ∈ {pk_1, ..., pk_ℓ}
10    K ←$ K
11    E ← E ⊎ {(pk_i, pk, c, K)}
12 return (c, K)

Oracle ADecap(j ∈ [ℓ], pk, c)
13 try get K s.t. (pk, pk_j, c, K) ∈ E
14    then return K
15 K ← AuthDecap(sk_j, pk, c)
16 return K
```

```
01 foreach i <= N do

02 s <-R keypairseed;
03 (
04 Opk() :=
05 return(pkgen(s))

06 |

07 foreach ie <= Qeperuser do
08 ks <-R kemseed; (
09 OAEncap(pk_R) :=
10 find i2 <= N suchthat defined(s[i2])
11    && pk_R = pkgen(s[i2]) then (
12  sk <- skgen(s);
13  let (k, ce) = AuthEncap_r(ks, pk_R, sk) in (
14    k' <-R key;
15    insert E(pkgen(s), pk_R, ce, k');
16    return(AuthEncap_tuple(k', ce))
17  ) else (
18    return(AuthEncap_None)
19  )
20 ) else
21  return(AuthEncap_r(ks, pk_R, skgen(s))))

22 |

23 foreach id <= Qdperuser do (
24 OADecap(pk_S, cd) :=
25 get E(=pk_S, =pkgen(s), =cd, k'') in (
26  return(k'')
27 ) else
28  return(AuthDecap(cd, skgen(s), pk_S)))

29 )
```

In pseudocode, L01 to L04 constitute the *main procedure* of the game; they set up the experiment and call the adversary, giving it the oracles Gen, AEncap and ADecap. AEncap and ADecap explicitly take the indices $i$ and $j$ as parameter, by which the adversary chooses the private key upon which it wants to act. These indices are assigned by Gen.

In CryptoVerif code, the first line R01 states that everything below will be executed in parallel N times. In CryptoVerif, this is called a *replication*. A replication index i is assigned to each of the parallel executions. Continuing to R02, in each of these parallel *processes*, a key pair will be generated. In our CryptoVerif model, we chose to represent key pairs in a general way by a key pair seed s, the private key computed from it by skgen(s) and the public key by pkgen(s). This allows to cover a broad range of KEM

constructions with one model. The type of key pair seed elements in our models is called `keypairseed`. The operator `<-R` denotes that the variable on the left is sampled randomly from the type given on the right. These keys can be dynamically generated, that is, line `R02` implicitly defines an oracle GenCV which generates a key pair seed `s`.

Line `R03` begins a block of oracle definitions, and it ends with line `R29`. In CryptoVerif, all oracles are implicitly accessible by the adversary. The three oracles that are defined are `Opk` in `R04`, `OAEncap` in `R09`, and `OADecap` in `R24`. Lines `R06` and `R22` define that these oracles are accessible by the adversary in parallel, in any order it chooses to call them.

The oracles GenCV, `Opk`, `OAEncap`, and `OADecap` are located *inside* the `N` times replication started in `R01`. This means that these oracles are available *once for each* $\mathtt{i} \in [1,\mathtt{N}]$, and they are defined with this index as implicit argument. Thus, the adversary has access to `N` oracles GenCV`[i]` for $1 \leq \mathtt{i} \leq \mathtt{N}$, and the variable defined in GenCV`[i]` is `s[i]`. All variables in CryptoVerif are implicitly stored in arrays indexed by the replication indices above their definition. Similarly, the adversary has access to `N` oracles `Opk[i]` for $1 \leq \mathtt{i} \leq \mathtt{N}$, and the variable accessed inside `Opk[i]` is `s[i]`. A call to GenCV`[i]` immediately followed by a call to `Opk[i]` is equivalent to a call to Gen, up to the renumbering of keys. (In pseudocode, the keys are indexed in increasing order. In CryptoVerif, the keys can be generated in any order. The renumbering just consists in applying a bijection known to the adversary to the indices of keys.) CryptoVerif separates generating keys by GenCV`[i]` from acquiring the public key by `Opk[i]`, while pseudocode groups these two actions in one oracle call Gen. However, a call to `Opk[i]` can be added immediately after GenCV`[i]` and all other calls to `Opk[i]` removed without changing the state of the system, so this separation leaves the security notion unchanged.

The main procedure in the pseudocode (lines `L01`–`L04`) is essentially absent in CryptoVerif. Line `L01` initializes the index for the last generated key $\ell$. There is no such index in CryptoVerif, since the adversary provides the index of the key it generates at generation time, as argument to oracle GenCV. Line `L02` initialises the multiset $\mathcal{E}$. In the CryptoVerif model, $\mathcal{E}$ is represented by a table `E`, which is implicitly empty at the beginning. Line `L03` calls the adversary and line `L04` returns its result. The adversary is not explicitly called in CryptoVerif: the adversary is implicit and is allowed to call all oracles provided by the CryptoVerif code.

Lines `R07` and `R23` define a replication, indicating that the adversary can call `OAEncap[i](pk_R)` for each key pair `Qeperuser` times, and `OADecap[i](pk_S, cd)` for each key pair `Qdperuser` times. The oracle `Opk` has no replication in front of it, which means the adversary can call it only once per key pair (which is sufficient to acquire the public key).

In the real version of the security notion, the oracle AEncap directly returns the result of the probabilistic AuthEncap function, see line `L08`. This corresponds to `R21`. In CryptoVerif, all functions are deterministic; to model a probabilistic function, one has to provide randomness to a function. For `AuthEncap_r` this is done with the `ks` parameter. It is chosen randomly by the oracle `OAEncap` independently for each call; the CryptoVerif syntax requires that this statement is located directly after the replication definition

in R08, and thus before the oracle definition.

The public key given as parameter to OAEncap has the role of the recipient public key, which is why it is called pk_R. We proceed to explain how the ideal version of the security notion is formalised in CryptoVerif. The if statement in L09 corresponds to the find statement started in lines R10 and R11. This find implements a lookup in the set of honest public keys $\mathrm{pk}_i$: it looks for an index i2 ∈ [1,N], and tests if the public key pk_R given as argument to the oracle is equal to the public key pkgen(s[i2]) corresponding to the key pair seed s[i2], that is, the value of s generated within the replicated copy indexed by i2. (The operator && is the logical and.)

The syntax let (...) = ... in R13 denotes a pattern matching; it is used here to split up the tuple that the function AuthEncap_r returns. The actual pattern matching used in the CryptoVerif model is let AuthEncap_tuple(k: key, ce: ciphertext) = ..., but we abbreviated it to let (k, ce) = ... such that it fits into one line in the listing. In general, a pattern matching can fail in CryptoVerif, which is why we need to include an else case with lines R17 and R18; we return an error symbol AuthEncap_None, which corresponds to ⊥. The actual AuthEncap does not fail; we model this with a simple game transformation eliminate_failing that expresses that AuthEncap_r always returns a pair and results in removing the else branch during the proofs. Lines R14 and R16 are executed if the pattern matching and thus the call to AuthEncap_r succeeded: A random KEM key k' is generated, a new element is added to the table E, which represents the multiset $\mathcal{E}$, and a new tuple is constructed from k' and the KEM ciphertext ce, and returned.

The difference remaining to be explained is that on the left side, there is only one call to AuthEncap, at the beginning of the oracle in line L08, but two calls to AuthEncap_r on the right side, in lines R13 and R21. These two calls are in disjoint branches of the code and thus both variants are equivalent. The reason we do not call AuthEncap_r once at the beginning of the oracle is purely technical: The proof is easier if we assign unique variable names per branch from the start (here: (k, ce) and no variable assignment in R21).

Oracle OADecap takes a sender public key pk_S and a KEM ciphertext cd as arguments. In the real version, the result of the actual AuthDecap function is returned, see lines L15 and R28.

The **try get** statement with the lookup in multiset $\mathcal{E}$ in L13 has its counterpart in the get statement in line R25, which looks up in table E. In E(=pk_S, =pkgen(s), =cd, k''), the equality tests =pk_S, =pkgen(s), and =cd indicate that we look for an element in E whose first 3 components are equal to the given values. The fourth component is a variable k'', which is bound to the fourth component of the found element of E, if such an element is found. If an element is found, OADecap returns the random key k'' found in the table E (R26). In case several elements match, the get statement chooses one at random, like the **try get** statement.

The CryptoVerif property expresses a bound on the probability that the adversary distinguishes the real from the ideal game; it leaves implicit the return of the bit $b$ by which the adversary says with which of the two games it thinks it interacts.

If CryptoVerif concludes a proof, it displays the probability bound it could prove. In CryptoVerif's output, this bound is indicated in the line that starts with `RESULT Proved`. The proof output for Chapter 4's theorems can be found in the files with filenames ending in `.proof` [Alw+]. The probability bound depends on replication parameters (for example `N`, `Qeperuser`, `Qdperuser`), total number of calls to oracles, written `#O` for oracle `O` (for example `#OAEncap`, `#OADecap`), probability bounds of the assumptions used in the proof (for example `Adv_PRF_KeySchedule` and `Adv_Outsider_CCA`), and collision probabilities (for example `P_pk_coll`, which corresponds to $P_{\mathsf{AKEM}}$, for AKEM public keys in our composition proofs, and `P_hashcoll` for the collision resistant hash function in the key schedule proof). Probability bounds like `Adv_Outsider_CCA` are expressed as functions with several arguments: the execution time of the adversary (indicated by `time_*`) and the numbers of queries. The execution time of the adversary is detailed in the lines below `RESULT Proved`.

[Alw+] Alwen et al., *Analysing the HPKE Standard Supplementary Material*

# D

# *cv2fstar Case Study: CryptoVerif Model and Generated Code*

## D.1 CryptoVerif Model nsl.ocv

```
(* Needham Schroeder public key protocol, fixed by Lowe.
   The user identity in this model is just an arbitrary
   address. An implementation could use an user@hostname
   construction.
 *)

param Qsetup.
param Qkey_reg.
param Qinit.
param Qresp.

def OptionType_1(option, option_Some, option_None, input) {
  fun option_Some(input): option [data].
  const option_None: option.
  equation forall x: input;
    option_Some(x) <> option_None.
}

type nonce [fixed,large].
type pkey [bounded].
type skey [bounded].
type keyseed [fixed,large].
type encseed [bounded].
type plaintext [bounded].
type address [bounded].
type keypair [bounded].
fun kp(pkey, skey): keypair [data].

type ciphertext [bounded].
type ciphertext_opt [bounded].
expand OptionType_1(ciphertext_opt, ciphertext_some, ciphertext_bottom, ciphertext).

table trusted_keys(address, skey, pkey).
table all_keys(address, pkey, bool).

set diffConstants = false.

fun msg1(nonce, address):plaintext [data].
fun msg2(nonce, nonce, address):plaintext [data].
fun msg3(nonce):plaintext [data].

```

```
42
43
44 equation forall z:nonce,t:nonce,u:address,y2:nonce,z2:address;
45   msg2(z,t,u) <> msg1(y2,z2).
46 equation forall y:nonce,y2:nonce,z2:address;
47   msg3(y) <> msg1(y2,z2).
48 equation forall z:nonce,t:nonce,u:address,y2:nonce;
49   msg2(z,t,u) <> msg3(y2).
50
51 (* Public-key encryption (IND-CCA2) *)
52
53 proba Penc.
54 proba Penccoll.
55
56 expand IND_CCA2_public_key_enc_all_args(
57   keyseed, pkey, skey, plaintext, ciphertext_opt, encseed,
58   skgen, skgen2, pkgen, pkgen2, enc, enc_r, enc_r2, dec_opt, dec_opt2, injbot,
59   Z, Penc, Penccoll).
60
61 (* Not needed because the in processes receive ciphertext, not ciphertext_opt *)
62 equation forall sk: skey; dec_opt(ciphertext_bottom, sk) = bottom.
63
64 letfun dec(c: ciphertext, sk: skey) =
65   dec_opt(ciphertext_some(c), sk).
66
67 letfun keygen() = k <-R keyseed; kp(pkgen(k), skgen(k)).
68
69 const Zplaintext: plaintext.
70 equation forall x: plaintext; Z(x) = Zplaintext.
71
72 (* Queries *)
73
74 event beginA(address, address, nonce, nonce).
75 event endA(address, address, nonce, nonce).
76 event beginB(address, address, nonce, nonce).
77 event endB(address, address, nonce, nonce).
78
79 query x:address, y:address, na:nonce, nb:nonce;
80   event(endA(x,y,na,nb)) ==> event(beginB(x,y,na,nb)).
81 query x:address, y:address, na:nonce, nb:nonce;
82   event(endB(x,y,na,nb)) ==> event(beginA(x,y,na,nb)).
83 query x:address, y:address, na:nonce, nb:nonce;
84   inj-event(endA(x,y,na,nb)) ==> inj-event(beginB(x,y,na,nb)).
85 query x:address, y:address, na:nonce, nb:nonce;
86   inj-event(endB(x,y,na,nb)) ==> inj-event(beginA(x,y,na,nb)).
87
88 type msg1res_t.
89 fun msg1succ(skey, pkey, bool, nonce, ciphertext): msg1res_t [data].
90 const msg1fail: msg1res_t.
91 equation forall x1: skey, x2: pkey, x3: bool, x4: nonce, x5: ciphertext;
92   msg1succ(x1,x2,x3,x4,x5) <> msg1fail.
93
94 implementation
95   type keyseed=256;
96   type nonce=64;
97   type ciphertext="ciphertext"
98     [serial = "serialize_ciphertext","deserialize_ciphertext"; equal = "eq_ciphertext"];
99   type ciphertext_opt="ciphertext_opt"
100    [equal = "eq_ciphertext_opt"; serial = "ciphertext_opt_to","ciphertext_opt_from"];
101  fun ciphertext_some="ciphertext_some" [inverse = "inv_ciphertext_some"];
102  const ciphertext_bottom="ciphertext_bottom";
103  type pkey="pkey" [serial = "serialize_pkey","deserialize_pkey"; equal = "eq_pkey"];
```

```
104  type skey="skey" [serial = "serialize_skey","deserialize_skey"; equal = "eq_skey"];
105  type encseed=256;
106  table trusted_keys="trusted_keys";
107  table all_keys="all_keys";
108  type keypair="keypair" [equal = "eq_keypair"];
109  type plaintext="plaintext" [equal = "eq_plaintext"];
110  fun msg1="msg1" [inverse = "inv_msg1"];
111  fun msg2="msg2" [inverse = "inv_msg2"];
112  fun kp = "kp" [inverse = "inv_kp"];
113  fun skgen = "skgen";
114  fun pkgen = "pkgen";
115  type address="address"
116    [serial = "serialize_address", "deserialize_address"; equal = "eq_addr"];
117  fun enc_r="enc";
118  fun dec="dec";
119  fun injbot="injbot" [inverse = "inv_injbot"];
120  fun msg3="msg3" [inverse = "inv_msg3"];
121  fun msg1succ="msg1succ" [inverse = "inv_msg1succ"];
122  const msg1fail="msg1fail";
123  type msg1res_t="msg1res" [equal = "eq_msg1res"].
124
125 letfun initiator_send_msg1_inner(addrA: address, addrX: address) =
126  (* the gets fail if addrA or addrX have not been
127     setup by the adversary. *)
128  get[unique] trusted_keys(=addrA, skA, pkA) in (
129    get[unique] all_keys(=addrX, pkX, trustX) in (
130      (* Prepare Message 1 *)
131      Na <-R nonce;
132      let cc1 = enc(msg1(Na, addrA), pkX) in
133      let ciphertext_some(c1: ciphertext) = cc1 in (
134        msg1succ(skA, pkX, trustX, Na, c1)
135      ) else msg1fail
136    ) else msg1fail
137  ) else msg1fail.
138
139 let initiator() =
140
141  Initiator {
142
143    foreach i_init ≤ Qinit do
144
145    initiator_send_msg1 (addrA: address, addrX: address) :=
146      let msg1succ(skA, pkX, trustX, Na, c1) = initiator_send_msg1_inner(addrA, addrX) in
147      return (c1);
148
149    (* Receive Message 2 *)
150    initiator_send_msg3 (c: ciphertext) :=
151      let injbot(msg2(=Na, Nb, =addrX)) = dec(c, skA) in
152      event beginA(addrA, addrX, Na, Nb);
153      (* Prepare Message 3 *)
154      let ciphertext_some(c3) = enc(msg3(Nb), pkX) in
155      return (c3);
156
157    (* OK *)
158    initiator_finish () :=
159      if (trustX) then
160      event endA(addrA, addrX, Na, Nb);
161      return ()
162
163  }.
164
165 let responder() =
```

```
166
167  Responder {
168
169  foreach i_resp ≤ Qresp do
170
171  (* Receive Message 1 *)
172  responder_send_msg2 (addrB: address, m: ciphertext) :=
173   (* the get fails if addrB has not been setup by
174      the adversary *)
175   get[unique] trusted_keys(=addrB, skB, pkB) in
176   let injbot(msg1(Na, addrY)) = dec(m, skB) in
177   get[unique] all_keys(=addrY, pkY, trustY) in
178   (* Send Message 2 *)
179   Nb <-R nonce;
180   event beginB(addrY, addrB, Na, Nb);
181   let ciphertext_some(c2) = enc(msg2(Na, Nb, addrB), pkY) in
182   return (c2);
183
184   (* Receive Message 3 *)
185   responder_receive_msg3 (m3: ciphertext) :=
186    let injbot(msg3(=Nb)) = dec(m3, skB) in
187    if (trustY) then (
188     event endB(addrY, addrB, Na, Nb); return ())
189    else return ()
190
191  }.
192
193 let key_register() =
194  Key_Register {
195
196  foreach i ≤ Qkey_reg do
197
198  register (addr: address, pkX: pkey) :=
199   get[unique] all_keys(=addr, ign1, ign2) in (
200    yield
201   ) else
202    insert all_keys(addr, pkX, false);
203    return ()
204  }.
205
206 let setup() =
207  Setup {
208
209   foreach i ≤ Qsetup do
210    setup(addr: address) :=
211     get[unique] all_keys(=addr, ign1, ign2) in (
212      yield
213     ) else
214      let kp(the_pkA: pkey, the_skA: skey) = keygen() in
215      insert trusted_keys(addr, the_skA, the_pkA);
216      insert all_keys(addr, the_pkA, true);
217      return(the_pkA)
218
219  }.
220
221 process
222  (
223   run setup()
224  |
225   run key_register()
226  |
227   run initiator()
```

```
  |
    run responder()
  )
```

## D.2 NSL.INITIATOR.FSTI, GENERATED BY CV2FSTAR

```
module NSL.Initiator

open CVTypes
open State
open Helper
open NSL.Types
open NSL.Functions
open NSL.Tables
open NSL.Sessions
open NSL.Events
open NSL.Protocol

val oracle_initiator_finish: nsl_state → nat → Tot (nsl_state * option (unit))

val oracle_initiator_send_msg_1: nsl_state → address → address
  → Tot (nsl_state * option (nat * ciphertext))

val oracle_initiator_send_msg_3: nsl_state → nat → ciphertext
  → Tot (nsl_state * option (ciphertext))
```

## D.3 NSL.INITIATOR.FST, GENERATED BY CV2FSTAR

```
module NSL.Initiator

open CVTypes
open State
open Helper
open NSL.Types
open NSL.Functions
open NSL.Tables
open NSL.Sessions
open NSL.Events
open NSL.Protocol


let oracle_initiator_finish state sid =
  match get_and_remove_session_entry state Oracle_initiator_finish sid with
  | state, None → state, None
  | state, Some (se: nsl_session_entry) →
    match se with
    | R1_initiator_finish var_Na_3 var_Nb_1 var_addrA_1 var_addrX_1 var_trustX_2 →
      if var_trustX_2
      then
        let ev = Event_endA var_addrA_1 var_addrX_1 var_Na_3 var_Nb_1 in
        let state = state_add_event state ev in
        (state, Some ())
```

```
    else state, None


let oracle_initiator_send_msg_1 state input_43 input_42 =
 let state, sid = state_reserve_session_id state in
 let var_addrA_1 = input_43 in
 let var_addrX_1 = input_42 in
 let state, v44 = NSL.Letfun.fun_initiator_send_msg_1_inner state var_addrA_1 var_addrX_1 in
 let bvar_45 = v44 in
 match inv_msg1succ bvar_45 with
 | None → state, None
 | Some (var_skA_2, var_pkX_4, var_trustX_2, var_Na_3, var_c1_3) →
  let sel =
   [R1_initiator_send_msg_3 var_Na_3 var_addrA_1 var_addrX_1 var_pkX_4 var_skA_2 var_trustX_2]
  in
  let state = state_add_to_session state sid sel in
  (state, Some (sid, var_c1_3))


let oracle_initiator_send_msg_3 state sid input_46 =
 match get_and_remove_session_entry state Oracle_initiator_send_msg_3 sid with
 | state, None → state, None
 | state, Some (se: nsl_session_entry) →
  match se with
  | R1_initiator_send_msg_3 var_Na_3 var_addrA_1 var_addrX_1 var_pkX_4 var_skA_2 var_trustX_2 →
   let var_c_3 = input_46 in
   let bvar_47 = (dec var_c_3 var_skA_2) in
   match inv_injbot bvar_47 with
   | None → state, None
   | Some bvar_48 →
    match inv_msg2 bvar_48 with
    | None → state, None
    | Some (bvar_49, var_Nb_1, bvar_50) →
     if eq_lbytes bvar_49 var_Na_3 && eq_addr bvar_50 var_addrX_1
     then
      let ev = Event_beginA var_addrA_1 var_addrX_1 var_Na_3 var_Nb_1 in
      let state = state_add_event state ev in
      let state, v51 = NSL.Letfun.fun_enc state (msg3 var_Nb_1) var_pkX_4 in
      let bvar_52 = v51 in
      match inv_ciphertext_some bvar_52 with
      | None → state, None
      | Some var_c3_0 →
       let sel =
        [R1_initiator_finish var_Na_3 var_Nb_1 var_addrA_1 var_addrX_1 var_trustX_2]
       in
       let state = state_add_to_session state sid sel in
       (state, Some (var_c3_0))
     else state, None
```

## D.4 NSL.RESPONDER.FSTI, GENERATED BY CV2FSTAR

```
module NSL.Responder

open CVTypes
open State
open Helper
open NSL.Types
```

```
open NSL.Functions
open NSL.Tables
open NSL.Sessions
open NSL.Events
open NSL.Protocol

val oracle_responder_receive_msg_3: nsl_state → nat → ciphertext
  → Tot (nsl_state * option (unit))

val oracle_responder_send_msg_2: nsl_state → address → ciphertext
  → Tot (nsl_state * option (nat * ciphertext))
```

## D.5 NSL.RESPONDER.FST, GENERATED BY CV2FSTAR

```
module NSL.Responder

open CVTypes
open State
open Helper
open NSL.Types
open NSL.Functions
open NSL.Tables
open NSL.Sessions
open NSL.Events
open NSL.Protocol


let oracle_responder_receive_msg_3 state sid input_18 =
  match get_and_remove_session_entry state Oracle_responder_receive_msg_3 sid with
  | state, None → state, None
  | state, Some (se: nsl_session_entry) →
    match se with
    | R1_responder_receive_msg_3 var_Na_4 var_Nb_2 var_addrB_0 var_addrY_0 var_skB_0 var_trustY_0 →
      let var_m3_0 = input_18 in
      let bvar_19 = (dec var_m3_0 var_skB_0) in
      match inv_injbot bvar_19 with
      | None → state, None
      | Some bvar_20 →
        match inv_msg3 bvar_20 with
        | None → state, None
        | Some bvar_21 →
          if eq_lbytes bvar_21 var_Nb_2
          then
            if var_trustY_0
            then
              let ev = Event_endB var_addrY_0 var_addrB_0 var_Na_4 var_Nb_2 in
              let state = state_add_event state ev in
              (state, Some ())
            else (state, Some ())
          else state, None


let oracle_responder_send_msg_2 state input_23 input_22 =
  let state, sid = state_reserve_session_id state in
  let var_addrB_0 = input_23 in
  let var_m_0 = input_22 in
  match
```

```
  get_unique state
   Table_trusted_keys
   (fun (te: nsl_table_entry Table_trusted_keys) →
      let TableEntry_trusted_keys tvar_26 tvar_25 tvar_24 = te in
      let bvar_27 = tvar_26 in
      let var_skB_0 = tvar_25 in
      let var_pkB_0 = tvar_24 in
      if eq_addr bvar_27 var_addrB_0 then Some (var_skB_0) else None)
with
| None → state, None
| Some var_skB_0 →
 let bvar_28 = (dec var_m_0 var_skB_0) in
 match inv_injbot bvar_28 with
 | None → state, None
 | Some bvar_29 →
  match inv_msg1 bvar_29 with
  | None → state, None
  | Some (var_Na_4, var_addrY_0) →
   match
    get_unique state
     Table_all_keys
     (fun (te: nsl_table_entry Table_all_keys) →
        let TableEntry_all_keys tvar_32 tvar_31 tvar_30 = te in
        let bvar_33 = tvar_32 in
        let var_pkY_0 = tvar_31 in
        let var_trustY_0 = tvar_30 in
        if eq_addr bvar_33 var_addrY_0 then Some (var_trustY_0, var_pkY_0) else None)
   with
   | None → state, None
   | Some (var_trustY_0, var_pkY_0) →
    let state, var_Nb_2 = call_with_entropy state (gen_lbytes 8) in
    let ev = Event_beginB var_addrY_0 var_addrB_0 var_Na_4 var_Nb_2 in
    let state = state_add_event state ev in
    let state, v34 =
     NSL.Letfun.fun_enc state (msg2 var_Na_4 var_Nb_2 var_addrB_0) var_pkY_0
    in
    let bvar_35 = v34 in
    match inv_ciphertext_some bvar_35 with
    | None → state, None
    | Some var_c2_0 →
     let sel =
      [
       R1_responder_receive_msg_3 var_Na_4
        var_Nb_2
        var_addrB_0
        var_addrY_0
        var_skB_0
        var_trustY_0
      ]
     in
     let state = state_add_to_session state sid sel in
     (state, Some (sid, var_c2_0))
```

## D.6 EXCERPT OF APPLICATION.FST, MANUALLY WRITTEN

```
val add_honest_parties: nsl_state → Tot (nsl_state * bool)
let add_honest_parties st =
```

```
 match oracle_setup st addrA with
 | st, None → st, false
 | st, Some pkA →
  match oracle_setup st addrB with
  | st, None → st, false
  | st, Some pkB → st, true

val initialize: Lib.RandomSequence.entropy → Tot (nsl_state * bool)
let initialize ent =
 let st = init_state nsl_state_type ent in
 add_honest_parties st

val testrun: Lib.RandomSequence.entropy → FStar.All.ML Lib.RandomSequence.entropy
let testrun ent =
 match initialize ent with
 | st0, false →
  IO.print_string "fail";
  finalize_state st0
 | st0, true →
  IO.print_string "## Tables After Calling Setup\n\n";
  nsl_print_tables st0;
  IO.print_string "\n";
  match oracle_initiator_send_msg_1 st0 addrA addrB with
  | st1, None →
   IO.print_string "init_send_1 fail";
   finalize_state st1
  | st1, Some (id_init, c1) →
   IO.print_string "## Sessions After init_send_msg_1\n\n";
   nsl_print_session st1 id_init;
   IO.print_string "\n\n";
   IO.print_string "## Protocol Message 1\n\n";
   print_label_bytes "ct" (serialize_ciphertext c1) true;
   IO.print_string "\n\n";
   match oracle_responder_send_msg_2 st1 addrB c1 with
   | st2, None →
    nsl_print_session st2 id_init;
    IO.print_string "resp_send_2 fail";
    finalize_state st2
   | st2, Some (id_resp, c2) →
    IO.print_string "## Sessions After resp_send_msg_2\n\n";
    nsl_print_session st2 id_init;
    nsl_print_session st2 id_resp;
    IO.print_string "\n\n";
    IO.print_string "## Protocol Message 2\n\n";
    print_label_bytes "ct" (serialize_ciphertext c2) true;
    IO.print_string "\n\n";
    match oracle_initiator_send_msg_3 st2 id_init c2 with
    | st3, None →
     nsl_print_session st3 id_init;
     IO.print_string "init_send_3 fail";
     finalize_state st3
    | st3, Some c3 →
     IO.print_string "## Sessions After init_send_msg_3\n\n";
     nsl_print_session st3 id_init;
     nsl_print_session st3 id_resp;
     IO.print_string "\n\n";
     IO.print_string "## Protocol Message 3\n\n";
     print_label_bytes "ct" (serialize_ciphertext c3) true;
     IO.print_string "\n\n";
     match oracle_responder_receive_msg_3 st3 id_resp c3 with
     | st4, None →
      nsl_print_session st4 id_resp;
```

```
         IO.print_string "resp_recv_3_fail";
         finalize_state st4
       | st4, Some () →
         IO.print_string "##_Sessions_After_resp_recv_msg_3\n\n";
         nsl_print_session st4 id_init;
         nsl_print_session st4 id_resp;
         IO.print_string "\n\n";
         match oracle_initiator_finish st4 id_init with
         | st5, None →
          IO.print_string "initiator_finish_fail";
          finalize_state st5
         | st5, Some () →
           IO.print_string "##_Sessions_After_init_finish\n\n";
           nsl_print_session st5 id_init;
           nsl_print_session st5 id_resp;
           IO.print_string "\n\n";
           IO.print_string "##_Event_List_at_the_End_of_Protocol_Execution\n\n";
           print_events st5 nsl_print_event;
           IO.print_string "\n\n";
           IO.print_string "success\n";
           finalize_state st5

let main =
 let entropy = testrun Lib.RandomSequence.entropy0 in
 ()
```

## D.7 OUTPUT OF APPLICATION.FST

```
## Tables After Calling Setup

trusted_keys [
{
 address: 41406c6f63616c686f7374,
 skey: 53cb277a99b5cb5e5415c70cba5253608e094fe241b30ab85b2d4c6a8f7eb12b,
 pkey: 6d7c8fe400c016674e62d84fab09ce7fd61b83d32324bf2825fd337d9d71a301,
}
{
 address: 42406c6f63616c686f7374,
 skey: d9f3f877c463f5a73615880de60f73dfe46df8a6dccc088306f602b1255a0141,
 pkey: fd91a8ca37dac95a1c17a766a4ee2461de2bb7f4da628cffe99e926eca380060,
}
]

all_keys [
{
 address: 41406c6f63616c686f7374,
 pkey: 6d7c8fe400c016674e62d84fab09ce7fd61b83d32324bf2825fd337d9d71a301,
 bool: ff,
}
{
 address: 42406c6f63616c686f7374,
 pkey: fd91a8ca37dac95a1c17a766a4ee2461de2bb7f4da628cffe99e926eca380060,
 bool: ff,
}
]
```

```
## Sessions After init_send_msg_1

0.R1_initiator_send_msg_3: {
  var_Na_3: 9bcae7aeb3fb1085,
  var_addrA_1: 41406c6f63616c686f7374,
  var_addrX_1: 42406c6f63616c686f7374,
  var_pkX_4: fd91a8ca37dac95a1c17a766a4ee2461de2bb7f4da628cffe99e926eca380060,
  var_skA_2: 53cb277a99b5cb5e5415c70cba5253608e094fe241b30ab85b2d4c6a8f7eb12b,
  var_trustX_2: ff,
}


## Protocol Message 1

  ct: adf4368b100ff6675c80ee04ed4667c5c88f489b72a3ddd50402b3089eccbe250cf857
     491f27ea9614bc0c5e326cb52f4f82703b6d5996df8643a75ada69bd7dccca06f43883,


## Sessions After resp_send_msg_2

0.R1_initiator_send_msg_3: {
  var_Na_3: 9bcae7aeb3fb1085,
  var_addrA_1: 41406c6f63616c686f7374,
  var_addrX_1: 42406c6f63616c686f7374,
  var_pkX_4: fd91a8ca37dac95a1c17a766a4ee2461de2bb7f4da628cffe99e926eca380060,
  var_skA_2: 53cb277a99b5cb5e5415c70cba5253608e094fe241b30ab85b2d4c6a8f7eb12b,
  var_trustX_2: ff,
}
1.R1_responder_receive_msg_3: {
  var_Na_4: 9bcae7aeb3fb1085,
  var_Nb_2: 3ac19b9d052e94f5,
  var_addrB_0: 42406c6f63616c686f7374,
  var_addrY_0: 41406c6f63616c686f7374,
  var_skB_0: d9f3f877c463f5a73615880de60f73dfe46df8a6dccc088306f602b1255a0141,
  var_trustY_0: ff,
}


## Protocol Message 2

  ct: d83c39ad7d4701a4d2462f9423667a32d88d64c31b27d8e326f325c5d5f89f09b67676a8494a80
     eaedf17e5741d01b619992fd27e0153980c6bad1f8277c43cb834c70391d0c697adde9f7a2acaf,


## Sessions After init_send_msg_3

0.R1_initiator_finish: {
  var_Na_3: 9bcae7aeb3fb1085,
  var_Nb_1: 3ac19b9d052e94f5,
  var_addrA_1: 41406c6f63616c686f7374,
  var_addrX_1: 42406c6f63616c686f7374,
  var_trustX_2: ff,
}
1.R1_responder_receive_msg_3: {
  var_Na_4: 9bcae7aeb3fb1085,
  var_Nb_2: 3ac19b9d052e94f5,
  var_addrB_0: 42406c6f63616c686f7374,
  var_addrY_0: 41406c6f63616c686f7374,
  var_skB_0: d9f3f877c463f5a73615880de60f73dfe46df8a6dccc088306f602b1255a0141,
  var_trustY_0: ff,
}

```

```
92
93 ## Protocol Message 3
94
95  ct: 48730c365da9350e69da4230e91bfb12db40b444177ada11ecdf8ef64
96     93b985557bd1cd4b3e99e647e0d82be740ed11e0c6771ead7ab9d0560,
97
98
99 ## Sessions After resp_recv_msg_3
100
101 0.R1_initiator_finish: {
102  var_Na_3: 9bcae7aeb3fb1085,
103  var_Nb_1: 3ac19b9d052e94f5,
104  var_addrA_1: 41406c6f63616c686f7374,
105  var_addrX_1: 42406c6f63616c686f7374,
106  var_trustX_2: ff,
107 }
108 1.session does not exist
109
110
111 ## Sessions After init_finish
112
113 0.session does not exist
114 1.session does not exist
115
116
117 ## Event List at the End of Protocol Execution
118
119 Events: [
120 beginB: { address: 41406c6f63616c686f7374,
121  address: 42406c6f63616c686f7374,
122  lbytes 8: 9bcae7aeb3fb1085,
123  lbytes 8: 3ac19b9d052e94f5,
124 }
125 beginA: { address: 41406c6f63616c686f7374,
126  address: 42406c6f63616c686f7374,
127  lbytes 8: 9bcae7aeb3fb1085,
128  lbytes 8: 3ac19b9d052e94f5,
129 }
130 endB: { address: 41406c6f63616c686f7374,
131  address: 42406c6f63616c686f7374,
132  lbytes 8: 9bcae7aeb3fb1085,
133  lbytes 8: 3ac19b9d052e94f5,
134 }
135 endA: { address: 41406c6f63616c686f7374,
136  address: 42406c6f63616c686f7374,
137  lbytes 8: 9bcae7aeb3fb1085,
138  lbytes 8: 3ac19b9d052e94f5,
139 }
140 ]
141
142 success
```

## RÉSUMÉ

Les protocoles cryptographiques sont l'un des fondements de la confiance que la société accorde aujourd'hui aux systèmes informatiques, qu'il s'agisse de la banque en ligne, d'un service web, ou de la messagerie sécurisée. Une façon d'obtenir des garanties théoriques fortes sur la sécurité des protocoles cryptographiques est de les prouver dans le modèle calculatoire. L'écriture de ces preuves est délicate : des erreurs subtiles peuvent entraîner l'invalidation des garanties de sécurité et, par conséquent, des failles de sécurité. Les assistants de preuve visent à améliorer cette situation. Ils ont gagné en popularité et ont été de plus en plus utilisés pour analyser des protocoles importants du monde réel, et pour contribuer à leur développement. L'écriture de preuves à l'aide de tels assistants nécessite une quantité substantielle de travail. Un effort continu est nécessaire pour étendre leur champ d'application, par exemple, par une automatisation plus poussée et une modélisation plus détaillée des primitives cryptographiques. Cette thèse montre sur l'exemple de l'assistant de preuve CryptoVerif et deux études de cas, que les preuves cryptographiques mécanisées sont praticables et utiles pour analyser et concevoir des protocoles complexes du monde réel.

La première étude de cas porte sur le protocole de réseau virtuel privé (VPN) libre et open source WireGuard qui a récemment été intégré au noyau Linux. Nous contribuons des preuves pour plusieurs propriétés typiques des protocoles de canaux sécurisés. En outre, nous étendons CryptoVerif avec un modèle d'un niveau de détail sans précédent du groupe Diffie-Hellman populaire Curve25519 utilisé dans WireGuard.

La deuxième étude de cas porte sur la nouvelle norme Internet « Hybrid Public Key Encryption » (HPKE), qui est déjà utilisée dans une extension du protocole TLS destinée à améliorer la protection de la vie privée (ECH), et dans Messaging Layer Security, un protocole de messagerie de groupe sécurisée. Nous avons accompagné le développement de cette norme dès les premiers stades avec une analyse cryptographique formelle. Nous avons fourni des commentaires constructifs ce qui a conduit à des améliorations significatives dans sa conception cryptographique. Finalement, nous sommes devenus un co-auteur officiel. Nous effectuons une analyse cryptographique détaillée de l'un des modes de HPKE, publiée à Eurocrypt 2021, un pas encourageant pour rendre les preuves cryptographiques mécanisées plus accessibles à la communauté des cryptographes.

La troisième contribution de cette thèse est de nature méthodologique. Pour des utilisations pratiques, la sécurité des implémentations de protocoles cryptographiques est cruciale. Cependant, il y a souvent un écart entre l'analyse de la sécurité cryptographique et l'implémentation, tous les deux basées sur la même spécification d'un protocole : il n'existe pas de garantie formelle que les deux interprétations de la spécification correspondent, et donc, il n'est pas clair si l'implémentation exécutable a les garanties prouvées par l'analyse cryptographique. Dans cette thèse, nous comblons cet écart pour les preuves écrites en CryptoVerif et les implémentations écrites en F*. Nous développons cv2fstar, un compilateur de modèles CryptoVerif vers des spécifications exécutables F* en utilisant la bibliothèque cryptographique vérifiée HACL* comme fournisseur de primitives cryptographiques. cv2fstar traduit les hypothèses non cryptographiques concernant, par exemple, les formats de messages, du modèle CryptoVerif vers des lemmes F*. Cela permet de prouver ces hypothèses pour l'implémentation spécifique, ce qui approfondit le lien formel entre les deux cadres d'analyse. Nous présentons cv2fstar sur l'exemple du protocole Needham-Schroeder-Lowe. cv2fstar connecte CryptoVerif au grand écosystème F*, permettant finalement de garantir formellement des propriétés cryptographiques sur du code de bas niveau efficace vérifié.



## MOTS CLÉS

protocoles cryptographiques, sécurité prouvée, méthodes formelles, implémentations vérifiées



## ABSTRACT

Cryptographic protocols are one of the foundations for the trust people put in computer systems nowadays, be it online banking, any web or cloud services, or secure messaging. One of the best theoretical assurances for cryptographic protocol security is reached through proofs in the computational model. Writing such proofs is prone to subtle errors that can lead to invalidation of the security guarantees and, thus, to undesired security breaches. Proof assistants strive to improve this situation, have got traction, and have increasingly been used to analyse important real-world protocols and to inform their development. Writing proofs using such assistants requires a substantial amount of work. It is an ongoing endeavour to extend their scope through, for example, more automation and detailed modelling of cryptographic building blocks. This thesis shows on the example of the CryptoVerif proof assistant and two case studies, that mechanized cryptographic proofs are practicable and useful in analysing and designing complex real-world protocols.

The first case study is on the free and open source Virtual Private Network (VPN) protocol WireGuard that has recently found its way into the Linux kernel. We contribute proofs for several properties that are typical for secure channel protocols. Furthermore, we extend CryptoVerif with a model of unprecedented detail of the popular Diffie-Hellman group Curve25519 used in WireGuard.

The second case study is on the new Internet standard Hybrid Public Key Encryption (HPKE), that has already been picked up for use in a privacy-enhancing extension of the TLS protocol (ECH), and in the Messaging Layer Security secure group messaging protocol. We accompanied the development of this standard from its early stages with comprehensive formal cryptographic analysis. We provided constructive feedback that led to significant improvements in its cryptographic design. Eventually, we became an official co-author. We conduct a detailed cryptographic analysis of one of HPKE's modes, published at Eurocrypt 2021, an encouraging step forward to make mechanized cryptographic proofs more accessible to the broader cryptographic community.

The third contribution of this thesis is of methodological nature. For practical purposes, security of implementations of cryptographic protocols is crucial. However, there is frequently a gap between a cryptographic security analysis and an implementation that have both been based on a protocol specification: no formal guarantee exists that the two interpretations of the specification match, and thus, it is unclear if the executable implementation has the guarantees proved by the cryptographic analysis. In this thesis, we close this gap for proofs written in CryptoVerif and implementations written in F*. We develop cv2fstar, a compiler from CryptoVerif models to executable F* specifications using the HACL* verified cryptographic library as backend. cv2fstar translates non-cryptographic assumptions about, e.g., message formats, from the CryptoVerif model to F* lemmas. This allows to prove these assumptions for the specific implementation, further deepening the formal link between the two analysis frameworks. We showcase cv2fstar on the example of the Needham-Schroeder-Lowe protocol. cv2fstar connects CryptoVerif to the large F* ecosystem, eventually allowing to formally guarantee cryptographic properties on verified, efficient low-level code.



## KEYWORDS

cryptographic protocols, provable security, formal methods, verified implementations