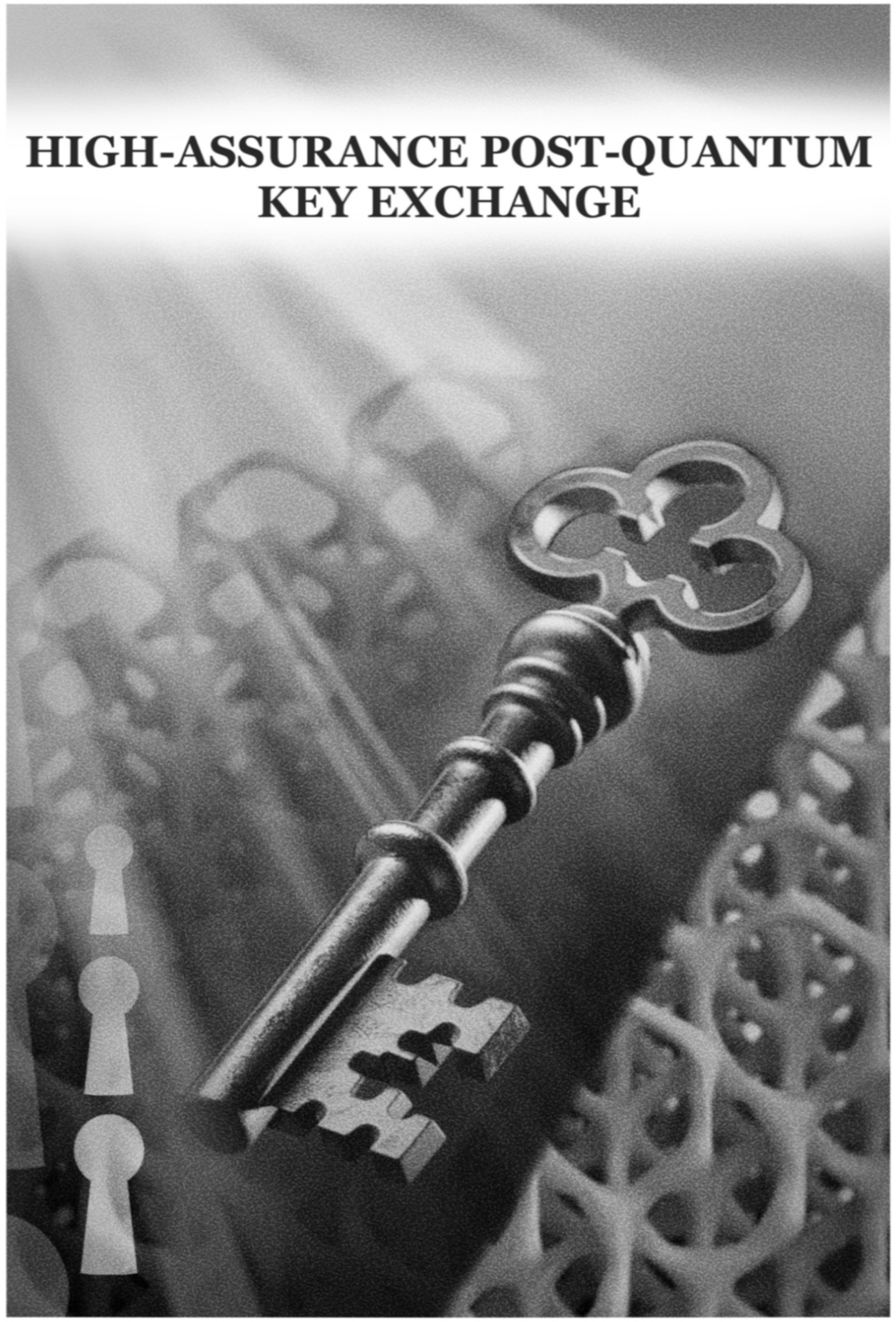
HIGH-ASSURANCE POST-QUANTUM KEY EXCHANGE

Miguel Quaresma



HIGH-ASSURANCE POST-QUANTUM KEY EXCHANGE

Miguel Quaresma

High-Assurance Post-Quantum Key Exchange

Miguel Quaresma

Copyright © Miguel Quaresma

Email: miguelquaresma.w@gmail.com
Website: https://www.mquaresma.dev/

First edition: 31st August 2026
Version: 31st August 2026

Printed by
Cover design: @Viki
"Trust, but verify" – Russian proverb

High-Assurance Post-Quantum Key Exchange

Proefschrift ter verkrijging van de graad van doctor aan de Radboud Universiteit Nijmegen op gezag van de rector magnificus prof. dr. J.M. Sanders, volgens besluit van het college voor promoties in het openbaar te verdedigen op

maandag 19 oktober 2026
om 10:30 uur precies

door

Miguel Miranda Quaresma
geboren op 23 juni 1997
te Évora, Portugal

Promotoren

Prof. dr. Peter Schwabe

Prof. dr. Gilles Barthe
Ruhr-Universität Bochum, Duitsland

Manuscriptcommissie

Prof. dr. Lejla Batina

Dr. Karthikeyan Bhargavan
Cryspen, Frankrijk

Prof. dr. Chitchanok Chuengsatiansup
Universität Potsdam, Duitsland

Prof. dr. Andreas Hülsing
Technische Universiteit Eindhoven

Prof. dr. Tamara Rezk
Chalmers Tekniska Högskola, Zweden

Acknowledgements

Firstly, I would like to thank my supervisors, Peter Schwabe and Gilles Barthe, for giving me the opportunity to explore exciting research topics and for their advice over the years. I am also thankful that you incentivized me to travel to different conferences, workshops, schools and research groups. These proved to be, not only professionally, but also personally valuable experiences that shaped my PhD.

I would also like to thank Chitchanok Chuengsatiansup, not only for being part of my reading committe, but for welcoming me in Melbourne and enabling me to explore a different research subject. I thank you, not only for the support and encouragement to pursue this project freely, but also for all the thoughtful discussions and meetings we had during that time.

I extend my appreciation to the other members of my reading committee, Joan Daemen, Karthikeyan Bhargavan, Andreas Hülsing, and Tamara Rezk, for taking the time to read my thesis and provide feedback.

I would also like to thank Goutam Tamvada, Santiago Olmos and Tiago Oliveira for your friendship, support and guidance, over the years. Thank you for making the never ending process of research and formal verification that much easier.

I would like to thank my coach, Matthias Graute. I know I wasn't the easiest the athlete to train in the last year but your unwavering support and honesty were fundamental, not only in sports but in life. Vielen Dank!

I also want to thank José Carlos Bacelar Almeida, my Master's supervisor. Your impact on my career trajectory cannot be overstated, the way you conduct your work, with integrity has always been inspiring. Obrigado!

I would like to thank my colleagues at Least Authority! The opportunity to start working with you at the tail end of my PhD is a privilege.

Obrigado aos meus amigos Eduardo Correia, João Vieira, José Martins, Simão Barbosa, João Pereira, pelo vosso apoio e amizade!

Obrigado Etienne Costa, sem ti nada disto teria acontecido. Obrigado por acreditares em mim!

Mais uma vez, agora em Português, obrigado Tiago Oliveira e Verónica Oliveira! A vossa companhia e os vossos conselhos foram uma casa longe de casa.

Noemi, thank you for making these last two years of my life very special. Thank you for sticking around, for your support and for your love! Grazie! Ti amo tanto!

Obrigado ao meu padrinho, Eduardo Street, pela tua companhia, pelo teu humor e por todas as nossas conversas.

Aos meus pais e à minha irmã, obrigado pelo apoio estes anos todos e por me terem proporcionado todas as oportunidades e ferramentas para chegar onde quisesse. Pai, obrigado por seres a minha referência e uma inspiração para mim, a minha admiração e gratitude por ti não tem limites. Mãe, obrigado por sempre saberes o que dizer, por me mostrares que a vida é mais 0s e 1s, mais do que um paper ou um doutoramento. Mana, obrigado pelo teu amor e apoio incondicional. Adoro-vos!

Finalmente, e acima de tudo, tenho que te agradecer a ti avô, Silvino Quaresma! O teu apoio incondicional, a tua crença em mim mesmo quando eu não a tinha em mim mesmo, foram fundamentais. Esta tese, dedico-a a ti!

Contents

Contents **vi**

1 Introduction **1**
1.1 Outline and Contributions 2
1.2 Data Management 5

2 Preliminaries **6**
2.1 Lattice-based Cryptography 6
2.1.1 Notation 7
2.1.2 Lattices 7
2.1.3 Learning with Errors (LWE) 8
2.1.4 Ring-LWE and Module-LWE 10
2.1.5 Security proofs and notions 12
2.2 KEM vs. NIKE 14
2.3 High-Assurance Implementations 15
2.3.1 Methodologies 16
2.3.2 Formosa Toolchain 19

3 Formally verifying Kyber – Episode IV: Implementation correctness **30**
3.1 Introduction 30
3.2 Specification of Kyber in EasyCrypt 35
3.2.1 Algebraic Structure 36
3.2.2 Algorithmic Sampling, Encoding and Decoding 41
3.3 Our toolchain: Jasmin and EasyCrypt 44
3.3.1 Jasmin before and after this paper 44
3.3.2 EasyCrypt and extensions added in this work 46
3.4 Implementation 48
3.5 Proof of functional correctness 51
3.5.1 Verifying the Reference Implementation 52
3.5.2 Verifying the AVX2 Implementation 56
3.5.3 Verifying the NTT 58
3.6 Benchmarks 59
3.7 Discussion 61

4 Formally verifying Kyber: Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt 64
4.1 Introduction . . . 65
4.2 Bird's Eye View of Development . . . 69
4.3 Preliminaries . . . 71
4.3.1 The EasyCrypt proof assistant . . . 71
4.3.2 Background on ML-KEM . . . 72
4.4 K-PKE correctness and IND-CPA security . . . 75
4.4.1 The Hashed Module LWE assumption . . . 76
4.4.2 The K-PKE construction. . . . 77
4.4.3 Derandomizing K-PKE . . . 81
4.5 Formalizing the FO Transform used in ML-KEM . . . 82
4.5.1 The T transform . . . 83
4.5.2 The $\mathsf{U}^{\not\perp}_m$ transform. . . . 86
4.5.3 The FO transform in ML-KEM . . . 90
4.6 Main result: Security of ML-KEM . . . 91
4.6.1 Instantiating FO_K with K-PKE . . . 91
4.6.2 Connection to the ML-KEM specification . . . 92
4.7 Connecting to ML-KEM implementations . . . 96

5 Swoosh: Efficient Lattice-Based Non-Interactive Key Exchange 99
5.1 Introduction . . . 100
5.1.1 NIKE vs. KEMs and Applications . . . 101
5.1.2 Our Contributions . . . 102
5.1.3 Related work . . . 103
5.2 Technical Outline . . . 104
5.3 Preliminaries . . . 107
5.3.1 Notation . . . 107
5.3.2 Quantum Preliminaries . . . 108
5.3.3 Hardness Assumption . . . 109
5.4 Definitions . . . 110
5.4.1 Non-Interactive Key Exchange . . . 110
5.5 Construction . . . 113
5.5.1 Passive Setting . . . 113
5.5.2 Active Setting . . . 118
5.5.3 Practical considerations . . . 119
5.5.4 Parameter selection . . . 120
5.6 Implementation & Performance Evaluation . . . 121
5.6.1 Implementation . . . 122
5.6.2 Performance and Security Evaluation . . . 124
5.7 Conclusions . . . 126

6 Jazzline: Composable CryptoLine functional-correctness proofs for Jasmin programs **127**
6.1 Introduction . . . 127
6.2 Motivating Example . . . 132
6.3 Annotated Jasmin Programs . . . 138
6.4 Certified compilation with traces . . . 139
6.5 Methodology . . . 141
6.6 Proving Partial Correctness using CryptoLine . . . 142
6.7 Trace Semantics in EasyCrypt . . . 146
6.8 Examples . . . 148
6.8.1 X25519 and X-Wing . . . 148
6.8.2 The ML-KEM NTT . . . 153
6.9 Conclusion . . . 159

7 Outlook **160**
7.1 Formosa toolchain . . . 161

Bibliography **162**

A Formally verifying Kyber: Episode V **187**
A.1 Cryptographic Definitions . . . 187
A.2 Generic EasyCrypt Arguments . . . 190
A.2.1 Plug-and-Pray . . . 191
A.2.2 Hybrids . . . 191
A.2.3 PROM . . . 191

B Swoosh: Efficient Lattice-Based Non-Interactive Key Exchange **193**
B.1 Proofs for Section 5.3 (Preliminaries) . . . 193
B.2 Proofs for Section 5.4 (Definitions) . . . 194
B.2.1 Non-Interactive Zero-Knowledge Proofs . . . 194
B.3 Proofs for Section 5.5 (Construction) . . . 197
B.3.1 Proof of Theorem 4 . . . 198

C Jazzline: Composable CryptoLine functional-correctness proofs for Jasmin programs **202**
C.1 Background on CryptoLine . . . 202

Summary **204**

Samenvatting **205**

About the Author **207**

1 Introduction

Key exchange mechanisms play a crucial role in modern cryptography, allowing parties to establish a shared secret that can be used to communicate securely. These mechanisms address the key distribution problem in secret-key cryptography: two or more parties must already share, or establish, a secret key before they can use secret-key primitives securely.

Broadly speaking, this asymmetry in how keys are handled constitutes the main difference between secret-key and public-key cryptography. Secret-key cryptography relies on a single (shared) secret key used by both parties for encryption, decryption and authentication, while public-key cryptography relies on a key pair composed of a private key and a public key. In this case, the private key is used to either decrypt or sign messages, and the public key is used to encrypt or verify a signature. Thus, the combination of public-key cryptography with secret-key cryptography is common: key exchange mechanisms, such as Non-Interactive Key Exchange (NIKE) and Key-Encapsulation Mechanisms (KEMs), allow two parties to establish a shared secret, which is then used to secure communications via secret-key primitives.

Traditionally, most of the public-key schemes have based their security on the hardness of a set of well known mathematical problems: factoring a large integer into its prime factors or computing discrete logarithms. Schemes such as RSA [184] and DH [83] rely on the hardness of these problems for their security. The recent advancement of quantum computers, capable of factoring large prime numbers or computing discrete logarithms in polynomial time, as opposed to super-polynomial time in classical computers, using Shor's algorithm [194] threaten the security of these schemes, commonly referred to as classical cryptography. Although there still exists no quantum computer large enough to break modern-day classical cryptography, the likelihood that such a system becomes a reality in the foreseeable future is increasing. This, coupled with threats such as harvest-now-decrypt-later, where encrypted data is captured for decryption at a future date when quantum computers are available, has motivated research into quantum-resistant cryptographic schemes, commonly referred to as post-quantum cryptography. The transition towards quantum-resistant cryptography requires a new set of mathematical problems

and structures that guarantee security against quantum computers. To address this threat, in January 2017, NIST announced a call for proposals to develop new, quantum-resistant cryptographic standards to replace classical ones, namely digital signature and key establishment schemes. As a result of the NIST PQC competition, in July 2022 and March 2025, a set of algorithms, encompassing both digital signatures and key-encapsulation mechanisms, have been selected for standardization: Dilithium [154], Falcon [180] and SPHINCS$^+$ [24] for the former, KYBER [26] and HQC [1] for the latter. The transition to post-quantum cryptography introduces a range of challenges, both technical and organizational in nature. The cause for these challenges range from the considerably different characteristics of the post-quantum primitives compared to classical counterparts, namely larger key sizes and ciphertexts, as well as the necessity to adapt protocol standards and devise deployment strategies for these primitives.

The field of computer-aided cryptography [35] explores the application of formal methods in cryptographic engineering in order to attain high-performance implementations with strong guarantees regarding attributes such as functional correctness and other properties such as memory safety and cryptographic constant-time [21]. Analogously to the migration to post-quantum cryptography, the adoption of high-assurance cryptography in real-world deployments has also faced resistance due to the challenges it poses, like the need to update or replace existing implementations. Despite that, high-assurance cryptography is already used in applications such as Firefox's NSS library [178, 51] and Google's BoringSSL library [94], which serve as examples of projects that attain strong guarantees of correctness and security through the use of formal methods. Even at a higher level, the security properties of protocols such as Signal's PQXDH (Post-Quantum Extended Diffie-Hellman) [54] and SPQR (Sparse Post Quantum Ratchet) [78, 52], the Noise framework [106], as well as Apple's iMessage PQ3 [146] have all been the subject of analysis via formal-methods tools in an attempt to obtain stronger security guarantees.

This thesis explores the utilization of formal methods for ensuring the security and functional correctness of cryptographic primitives and their implementations, respectively. In particular, it focuses on the development and application of formal methods to deployment-ready implementations of post-quantum key exchange schemes, using a high-assurance implementation of ML-KEM [167], and the KYBER scheme from which it was derived, as the main case study.

1.1 Outline and Contributions

All the work and results presented in this thesis are the result of collaborations with several co-authors. As is common in the cryptographic field, authors are listed in alphabetical order. The following section provides an outline of this thesis and an overview of the underlying publications, as well as my

contributions.

The work contained in Chapter 3 presents the first formally verified implementation of a post-quantum cryptographic scheme, Kyber, using the Jasmin framework and EasyCrypt proof assistant. The paper discusses an approach to produce reference and optimized AMD64 implementations of Kyber using Jasmin and then use EasyCrypt to write correctness proofs that connect these implementations to a handwritten (EasyCrypt) specification of the scheme. This work was extended in Chapter 4 to include a proof of IND-CCA security for ML-KEM, the Kyber-based KEM chosen for standardization by NIST. This later work provides machine-checked security proofs of ML-KEM in EasyCrypt, including IND-CPA security proofs of Kyber, a formalization of the Fujisaki-Okamoto transform and an IND-CCA security proof of ML-KEM. It connects these proofs to variants of the Jasmin implementations in Chapter 3, thereby providing an end-to-end security proof of an optimized implementation of ML-KEM [167].

> José Bacelar Almeida and Manuel Barbosa and Gilles Barthe and Benjamin Grégoire and Vincent Laporte and Jean-Christophe Léchenet and Tiago Oliveira and Hugo Pacheco and Miguel Quaresma and Peter Schwabe and Antoine Séré and Pierre-Yves Strub. "Formally verifying Kyber – Episode IV: Implementation correctness". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (June 2023), pp. 164–193. DOI: 10.46586/tches.v2023.i3.164-193

> José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe and Pierre-Yves Strub. "Formally Verifying Kyber - Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt". In: *Advances in Cryptology – CRYPTO 2024, Part II*. ed. by Leonid Reyzin and Douglas Stebila. Vol. 14921. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2024, pp. 384–421. DOI: 10.1007/978-3-031-68379-4_12

These two publications are the result of a large effort aiming to produce an end-to-end security proof of an optimized implementation of ML-KEM/Kyber. Consequently, contributors to this project are involved in both publications. The majority of my contributions pertain to the work delineated in Chapter 3, where I wrote large parts of the reference implementation and the whole optimized implementation in Jasmin of Kyber (and parts of ML-KEM) as well as the correctness proofs for the optimized implementation, excluding most of the NTT proof, in EasyCrypt. In addition, I also contributed

to add support for several new instructions in Jasmin and EasyCrypt required by the aforementioned implementations. All authors contributed to the writing of the paper. My contributions to the work in Chapter 4 are restricted to technical discussions, paper writing and the process of proofreading.

The work in Chapter 5 explores the feasibility of using lattices to build an efficient NIKE. The work provides an efficient lattice-based NIKE by first building a passively secure scheme and then converting it, through a generic compiler, to an actively-secure NIKE. The efficiency of the scheme is demonstrated via an optimized implementation of a concrete instantiation of the passively-secure construction, using Rust and Jasmin.

> Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta and Peter Schwabe. "SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange". In: *USENIX Security 2024: 33rd USENIX Security Symposium*. Ed. by Davide Balzarotti and Wenyuan Xu. Philadelphia, PA, USA: USENIX Association, Aug. 2024. URL: `https://www.usenix.org/conference/usenixsecurity24/presentation/gajland`

In this work I contributed by collaborating on the security analysis of the construction when instantiated with concrete parameters as well as by writing the optimized implementation in Rust and Jasmin. Additionally, I conducted comparative performance benchmarks with other NIKEs and KEMs to analyse the applicability of the proposed construction. All authors contributed to the writing of the paper.

The work in Chapter 6 extends the Jasmin framework to support automatic translation into CryptoLine models. This work provides a way to automatically prove subgoals in functional-correctness proofs of Jasmin programs. It also extends EasyCrypt to allow for the automatic integration of CryptoLine proofs to EasyCrypt. It demonstrates the applicability of this approach by providing the first formally verified implementation of the X-Wing hybrid KEM.

> José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Lionel Blatter, Gustavo Delerue, João Diogo Duarte, Benjamin Gregoire, Tiago Oliveira, Miguel Quaresma, Pierre-Yves Strub, Ming-Hsien Tsai, Bow-Yaw Wang and Bo-Yin Yang. "Jazzline: Composable CryptoLine Functional Correctness Proofs for Jasmin Programs". In: *ACM CCS 2025: 32nd Conference on Computer and Communications Security*. ACM Press, Nov. 2025, pp. 1409–1423. DOI: `10.1145/3719027.3744814`

My contribution to this work was conception of the initial idea behind Jazzline and the implementation of the translation logic of vectorized Jasmin

code into CryptoLine models. Additionally, I also modified the vectorized Jasmin implementation of the Kyber NTT and annotated it to allow extracting it to CryptoLine. All authors contributed to the writing of the paper.

1.2 Data Management

The research datasets that have been produced during this PhD research are available online, in a single archive, at: `https://doi.org/10.5281/zenodo.18421727`. Each individual research dataset is also available online on GitHub:

Chapter 3 Formally verifying Kyber – Episode IV: Implementation correctness
`https://github.com/formosa-crypto/formosa-mlkem`

Chapter 4 Formally verifying Kyber: Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt
`https://github.com/formosa-crypto/formosa-mlkem`

Chapter 5 Swoosh: Efficient Lattice-Based Non-Interactive Key Exchange
`https://github.com/MQuaresma/pswoosh`

Chapter 6 Jazzline: Composable CryptoLine functional-correctness proofs for Jasmin programs
`https://github.com/MQuaresma/jasmin/tree/cryptoline-mc`

2
Preliminaries

This chapter presents the most relevant background about lattice-based cryptography, the differences between NIKEs and KEMs, and high-assurance cryptographic implementations.

2.1 Lattice-based Cryptography

Lattice-based cryptography dates back to Ajtai's seminal work [2], which introduced worst-case-to-average-case reductions for lattice problems. In particular, it showed that solving random instances of the Small Integer Solution (SIS) problem is at least as hard as solving certain worst-case lattice problems, such as the Shortest Vector Problem (SVP). Later, Regev introduced the Learning with Errors (LWE) problem [182] and showed, via a quantum reduction, that solving average-case LWE is at least as hard as solving certain worst-case lattice problems. The LWE problem has become one of the central hardness assumptions underlying modern lattice-based cryptography. Although these works provided strong foundations for lattice-based cryptography, early constructions based on unstructured lattice assumptions were often inefficient.

The development of structured lattice variants, such as Ring-LWE [152] and Module-LWE [144], mitigated these efficiency limitations of early lattice-based constructions and showed that lattice-based cryptography could be efficient by enabling compact representations and fast arithmetic.

This tendency towards the development of efficient, lattice-based, quantum-secure schemes experienced a notable acceleration in the 2000s, largely driven by advancements in the field of quantum computing aimed at solving problems that would break classical cryptography. Ultimately, the lattice-based schemes submitted as part of NIST's endeavor to standardize post-quantum cryptosystems in 2017 [170] demonstrated respectable performance, even improving on some classical counterparts.

The two lattice-based schemes discussed in this thesis, Kyber and Swoosh, are based on the hardness of variants of the LWE problem [182]. Therefore, the following sections will provide a short background on this topic.

2.1.1 Notation

This section introduces some notation used throughout the preliminary chapter of this thesis.

For any positive integer $\beta \in \mathbb{Z}$, $[\beta]$ defines the set $[\beta] \coloneqq \{-\beta, \ldots, -1, 0, 1, \ldots, \beta\}$. This notation can be extended for vectors and matrices with coefficients in this set as $[\beta]^{n \times m}$.

Bold upper-case letters $\mathbf{A}$ and bold lower case letters $\mathbf{v}$ define matrices and vectors respectively. The Euclidean norm of a vector $\mathbf{v} = (x_0, x_1, ..., x_{n-1})$ is defined as the square root of the sum of its coordinates: $\|\mathbf{v}\| = \sqrt{x_0^2 + x_1^2 + ... + x_{n-1}^2}$. It represents the distance from the origin to the point represented by the vector's coordinates, i.e. the vector's length.

Given a set $\mathcal{S}$, $s \leftarrow \mathcal{S}$ denotes that s is uniformly sampled from this set. Analogously, this notation can be extended to vectors and matrices, $\mathbf{s} \leftarrow \mathcal{S}^{n \times m}$ with $\mathbf{s} \in \mathbb{Z}^{n \times m}$ to denote that each element of $\mathbf{s}$ is uniformly sampled from $\mathcal{S}$.

Let $\mathbb{Z}_q$ denote the set of integers from 0 up to $q-1$ and $(\mathbb{Z}_q, +, \times)$ denote the ring of integers modulo q. For polynomial rings, $\mathbb{Z}[x]$ is used to denote the polynomial ring with variable x and integer coefficients; $\mathcal{R}_q = \mathbb{Z}_q[x]$ is used to denote a polynomial ring with coefficients in $\mathbb{Z}_q$. Finally, $\mathcal{R}_{q,f} = \mathbb{Z}_q[x]/f(x)$ denotes the polynomial ring $\mathcal{R}_q$ where operations are performed modulo the polynomial $f(x)$ of degree d. For lattices, $\lambda(\mathcal{L})$ will denote the length of the shortest non-zero vector in the lattice $\mathcal{L}$, as determined by the Euclidean norm.

2.1.2 Lattices

As the name implies, lattices are the geometric objects at the base of lattice-based cryptography. An m-dimensional integer lattice is a discrete set of points composed by the integer linear combinations of n linearly independent vectors $(m \geq n)$, called a basis.

Definition 1. Formally, an integer lattice $\mathcal{L}(B)$ is a subgroup of $(\mathbb{Z}^m, +)$, and can be described via its basis

$$\mathbf{B} = (\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n) \in \mathbb{R}^m$$

as the following set

$$\mathcal{L}(B) = \{\mathbf{v} \in \mathbb{Z}^m : \exists \mathbf{z} \in \mathbb{Z}^m \ s.t. \ \mathbf{Bz} = \mathbf{v}\}$$

Lattice-based cryptographic constructions use a special type of lattices called q-ary integer lattices, where vectors are in $\mathbb{Z}_q^m$ as opposed to $\mathbb{Z}^m$. In addition to the improved efficiency that comes with bounding the vector elements to q, instead of working with arbitrarily large integers, the worst-case-to-average-case reduction in [182] shows that solving a certain problem over a random q-ary lattice is as hard as solving a certain problem for any lattice. As a result, basing cryptographic constructions on q-ary lattices preserves the security provided by hard lattice problems while allowing the instantiation of practical schemes.

2.1.2.1 Computational problems in lattices

There are several computational problems related to lattices that can be used to build cryptographic primitives. The following two fundamental problems, and their respective variants, are at the core of most modern lattice-based cryptography.

Definition 2 (Shortest Vector Problem - SVP). Given a basis $\mathbf{B}$ for a lattice $\mathcal{L}(\mathcal{B})$, find a shortest non-zero vector $v \in \mathcal{L}$ s.t. it minimizes the Euclidean norm $||v|| = \lambda(\mathcal{L})$.

Definition 3 (Closest Vector Problem - CVP). Given a basis $\mathbf{B}$ for a lattice $\mathcal{L}(\mathcal{B})$ and a vector $w \in \mathcal{R}^m$ such that w is not in $\mathcal{L}$, compute a vector $v \in \mathcal{L}$ that is the closest to w, i.e. it minimizes the Euclidean norm $||w - v||$.

In general, both problems become computationally hard for large lattice dimensions. This hardness is at the core of many lattice-based cryptographic schemes. A variant of SVP, called Approximate SVP, where the goal is finding an approximate solution to the SVP problem based on an approximation factor $\gamma \geq 1$[1], is of particular relevance for lattice-based cryptography:

Definition 4 (Approximate Shortest Vector Problem - SVP$_\gamma$). Given a basis $\mathbf{B}$ for a lattice $\mathcal{L}(\mathcal{B})$ of dimension n, find a short non-zero vector $v \in \mathcal{L}$ s.t. $||v|| = \gamma(n) \cdot \lambda(\mathcal{L})$.

Cryptographic schemes based on lattices, whether on SIS or LWE related problems, have reductions from the aforementioned computational problems in lattices.

2.1.3 Learning with Errors (LWE)

In [182], Regev set a foundation upon which many modern lattice schemes are based on, such as Swoosh and KYBER, by introducing the LWE problem. Regev showed that solving a random instance of the LWE problem was as hard as solving worst-case lattice problems and provided a scheme based on the hardness of LWE. Informally, the fundamental premise underlying the LWE problem asserts that, while solving a system of equations modulo q is easy (e.g. using Gaussian elimination), the introduction of noise via a small error makes this problem computationally hard. This hardness assumption can be illustrated by means of the following decision problem.

Definition 5 (Decision-LWE problem). Given n, q, and χ an error distribution in Z_q, the Decision-LWE problem consists of deciding which of the following two distributions generated a pair of values:

- $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$, with $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$, $\mathbf{s} \xleftarrow{\$} \mathbb{Z}_q^n$ and $e \leftarrow \chi$

[1] Usually based on the dimension of the lattice n.

- $(\mathbf{a}, u)$, with $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$ and $u \xleftarrow{\$} \mathbb{Z}_q$

The hardness of the LWE problem is based on the dimension n, the modulus q, the secret distribution, and the error distribution χ. In particular, the relation between the size of the modulus q, and the size of the error e plays an important role.

It is worth noting that in the standard formulation of LWE, the secret vector $\mathbf{s}$ is sampled uniformly from $\mathbb{Z}_q^n$. However, many practical lattice-based schemes sample the secret from a small distribution, often the same type of distribution used for the error term. This variant is commonly referred to as small-secret LWE and will be used in the following examples.

2.1.3.1 LWE Encryption Scheme

Based on the work in [182], the authors introduce an LWE-based public-key encryption scheme. A public-key encryption scheme can be defined as the following algorithm triple: key generation, encryption and decryption.

Algorithm 1 Key generation

Ensure: Secret key $sk \in \mathbb{Z}_q^n$, Public key $pk \in \mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n$

1: $\mathbf{s} \leftarrow \chi^n$
2: $\mathbf{e} \leftarrow \chi^n$
3: $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times n}$
4: $\mathbf{t} \leftarrow \mathbf{A}\mathbf{s} + \mathbf{e}$
5: $sk := \mathbf{s}$
6: $pk := (\mathbf{A}, \mathbf{t})$
7: **return** (sk, pk)

Algorithm 2 Encryption

Require: Message $b \in \{0, 1\}$, Public key $pk \in \mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n$
Ensure: Ciphertext $c \in \mathbb{Z}_q^n \times \mathbb{Z}_q$

1: $(\mathbf{A}, \mathbf{t}) := pk$
2: $\mathbf{r} \leftarrow \chi^n$
3: $\mathbf{e_1} \leftarrow \chi^n$
4: $e_2 \leftarrow \chi$
5: $\mathbf{u}^T \leftarrow \mathbf{A}^T + \mathbf{r} + \mathbf{e_1}$
6: $v \leftarrow \mathbf{t}^T \mathbf{r} + e_2 + \lfloor \frac{q}{2} \rceil b$
7: $c \leftarrow (\mathbf{u}, v)$
8: **return** c

Algorithm 3 Decryption

Require: Ciphertext $c \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, Secret key $sk \in \mathbb{Z}_q^n$
Ensure: Message $b \in \{0,1\}$
1: $\mathbf{s} := sk$
2: $(\mathbf{u}, v) := c$
3: $w \leftarrow v - \mathbf{u}^T\mathbf{s}$
4: $b \leftarrow round(w)$
5: **return** b

In this scheme, distinguishing the public key $(\mathbf{A}, \mathbf{As}+\mathbf{e})$ from $(\mathbf{A}, \mathbf{u})$ where $\mathbf{u}$ is random, i.e. sampled from a uniform distribution, can be seen as an instantiation of the (small secret) LWE hardness assumption. The public key can be used to encrypt a bit message b, and the private key can be used to decrypt the resulting ciphertext. The decryption algorithm yields a residual error value, $\mathbf{r}^T\mathbf{e} + e_2 - \mathbf{e}_1^T\mathbf{s}$, in addition to the original plaintext bit. Given the magnitude of each coefficient in the error and secret vectors, this additional error is at most $2n\beta^2 + \beta$. Consequently, if the parameters are selected such that $2n\beta^2 + \beta < \frac{q}{4}$, the decryption function is able to determine whether w is closer to 0 or $\frac{q}{2}$, and recover b using the *round* function.

2.1.4 Ring-LWE and Module-LWE

A clear limitation of the previous scheme is the inability to encrypt more than one bit of plaintext at a time, despite the ciphertext having a size of $(n+1)\log q$ bits, i.e. linear in the size of the original plaintext. One potential solution to the ciphertext expansion problem is to utilize larger public keys. Assuming a message length of $l \times k$ bits, it is possible to modify the previous scheme to encrypt such a message using larger keys, as shown in Algorithms 4 and 5[2].

Algorithm 4 Key generation

Ensure: Secret key $sk \in \mathbb{Z}_q^{n\times l}$, Public key $pk \in \mathbb{Z}_q^{n\times n} \times \mathbb{Z}_q^{n\times l}$
1: $\mathbf{S} \leftarrow \chi^{n\times l}$
2: $\mathbf{E} \leftarrow \chi^{n\times l}$
3: $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n\times n}$
4: $\mathbf{T} \leftarrow \mathbf{AS} + \mathbf{E}$
5: $sk := \mathbf{S}$
6: $pk := (\mathbf{A}, \mathbf{T})$
7: **return** (sk, pk)

[2]The decryption algorithm is identical and is omitted for simplicity.

Algorithm 5 Encryption

Require: Message $\mathbf{M} \in \{0,1\}^{l \times k}$, Public key $pk \in \mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^{n \times l}$
Ensure: Ciphertext $C \in \mathbb{Z}_q^{n \times k} \times \mathbb{Z}_q^{l \times k}$
1: $(\mathbf{A}, \mathbf{T}) \coloneqq pk$
2: $\mathbf{R} \leftarrow \chi^{n \times k}$
3: $\mathbf{E_1} \leftarrow \chi^{n \times k}$
4: $\mathbf{E_2} \leftarrow \chi^{l \times k}$
5: $\mathbf{U} \leftarrow \mathbf{A}^T\mathbf{R} + \mathbf{E_1}$
6: $\mathbf{V} \leftarrow \mathbf{T}^T\mathbf{R} + \mathbf{E_2} + \lfloor \frac{q}{2} \rceil \mathbf{M}$
7: $C \leftarrow (\mathbf{U}, \mathbf{V})$
8: **return** C

The resulting public key is $(n \times (n + l)) \log q$ bits in size, as opposed to $n \times (n + 1)$ bits in Algorithm 1[3]. However, the ciphertext is still considerably larger than the original plaintext, containing an additional $(k \times n) \log q$ bits.

An alternative and more efficient approach, proposed in [152], introduced a variant of LWE based on polynomial rings, called Ring-LWE (RLWE), which addressed the lack of efficiency of the former by performing the ring operations in a polynomial ring $\mathcal{R}_{q,f}$. Definition 6 presents the Decision-RLWE problem, the analogue of the Decision-LWE problem in the Ring-LWE setting.[4]

Definition 6 (Decision-RLWE problem). Given d, q, the ring $\mathcal{R}_{q,f}$, and χ an error distribution in $\mathcal{R}_{q,f}$, the Decision-RLWE problem consists of deciding which of the following two distributions generated a pair of values:

- $(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + \mathbf{e})$, with $\mathbf{a} \xleftarrow{\$} \mathcal{R}_{q,f}$, $\mathbf{s} \xleftarrow{\$} \mathcal{R}_{q,f}$ and $\mathbf{e} \leftarrow \chi$
- $(\mathbf{a}, \mathbf{u})$, with $\mathbf{a} \xleftarrow{\$} \mathcal{R}_{q,f}$ and $\mathbf{u} \xleftarrow{\$} \mathcal{R}_{q,f}$

RLWE can serve as the foundation for an encryption scheme that addresses the aforementioned limitations, permitting messages of d bits.

Although RLWE allows for improved efficiency, as a consequence of the added algebraic structure inherited from the polynomial rings, the security reductions from RLWE to hard lattice problems, such as SVP_γ, are restricted to the special class of ideal lattices. On the other hand, although LWE has been shown to be at least as hard as worst-case lattice problems, plain LWE-based constructions often have larger keys/ciphertexts than structured variants. Module-LWE, introduced in [144], addresses this gap between security and efficiency in LWE-based constructions by working over modules of rank k over

[3] One common optimization includes using a seed to generate the random distribution used to sample the matrix $\mathbf{A}$, and storing this seed instead of the full matrix.

[4] In the original paper [152], the authors presented a search version of the RLWE problem and a reduction to the decision counterpart under certain parameters. This counterpart is presented here.

a ring $\mathcal{R}_{q,f}$. The analogue of the decision problem in MLWE can be defined as follows:

Definition 7 (Decision-MLWE problem). Given q, d, a rank k, the ring $\mathcal{R}_{q,f}$, and χ an error distribution in $\mathcal{R}_{q,f}$, the Decision-MLWE problem consists of deciding which of the following two distributions generated a pair of values:

- $(\mathbf{A}, \mathbf{As} + \mathbf{e})$, with $\mathbf{A} \xleftarrow{\$} \mathcal{R}_{q,f}^{k\times k}$, $\mathbf{s} \xleftarrow{\$} \mathcal{R}_{q,f}^{k}$ and $\mathbf{e} \leftarrow \chi^k$
- $(\mathbf{A}, \mathbf{u})$, with $\mathbf{A} \xleftarrow{\$} \mathcal{R}_{q,f}^{k\times k}$ and $\mathbf{u} \xleftarrow{\$} \mathcal{R}_{q,f}^{k}$

This is the basis of the KYBER CPA scheme, and the ML-KEM standard [167], which operate on the polynomial ring $\mathcal{R}_{3329,X^{256}+1}$, with the rank k ranging from 2 for KYBER512 to 4 in the case of KYBER1024. In addition, SWOOSH is also based on the hardness of MLWE.

2.1.5 Security proofs and notions

As part of the design of cryptographic schemes, it is standard practice to assess the security of said schemes in relation to a specific adversary whose capabilities are well-defined.

In the context of public-key encryption schemes, these assessments, commonly referred to as "security proofs", primarily utilize three well-known security notions that differ based on the capabilities of the adversaries.

Consider $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ a public-key encryption scheme, the following definitions formalize the security notions associated with such a scheme. Consider, as well, a Probabilistic Polynomial-Time (PPT) adversary $\mathcal{A}$, to model algorithms that can potentially break the scheme being analysed.

Definition 8 (One-Way under Chosen-Plaintext Attack (OW-CPA)). A PKE scheme is one-way under chosen-plaintext attacks if, for all PPT adversaries $\mathcal{A}$, the adversary advantage ($\mathsf{Adv}^{\mathsf{OW\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) = |\Pr[\mathsf{OW\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE}} \Rightarrow \mathit{true}]|$) is negligible in the following game:

$\underline{\mathsf{OW\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE}}}$

1: $(pk, sk) \leftarrow_{\$} \mathsf{Gen}()$
2: $m \leftarrow_{\$} \mathcal{M}$
3: $c \leftarrow \mathsf{Enc}(pk, m)$
4: $m' \leftarrow_{\$} \mathcal{A}(pk, c)$
5: **return** $m' = m$

A scheme is considered $\mathsf{OW\text{-}CPA}$-secure if an adversary in possession of only the public key, is unable to recover the original message when presented with a random challenge ciphertext.

Definition 9 (Indistinguishability under Chosen-Plaintext Attack (IND-CPA)). A PKE scheme is indistinguishable under chosen-plaintext attacks if, for all PPT adversaries $\mathcal{A}$, the adversary advantage $(\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) = |\Pr[\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE}} \Rightarrow \mathit{true}] - 1/2|)$ is negligible in the following game:

$\underline{\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE}}}$

1: $(pk, sk) \leftarrow\!\!\$\ \mathsf{Gen}()$
2: $(m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}(pk)$
3: $b \leftarrow \{0, 1\}$
4: $c_b \leftarrow \mathsf{Enc}(pk, m_b)$
5: $b' \leftarrow\!\!\$\ \mathcal{A}(pk, c_b)$
6: **return** $b' = b$

In IND-CPA, the adversary is allowed to request the encryption of arbitrary messages. A scheme is secure in this setting if the adversary is unable to distinguish which of the two adversary-chosen plaintexts was encrypted.

Definition 10 (Indistinguishability under adaptive Chosen-Ciphertext Attack (IND-CCA)[5]). A PKE scheme is indistinguishable under adaptive chosen-ciphertext attacks if, for all PPT adversaries $\mathcal{A}$, the adversary advantage $(\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}}(\mathcal{A}) = |\Pr[\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{PKE}} \Rightarrow \mathit{true}] - 1/2|)$ is negligible in the following game:

$\underline{\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{PKE}}}$

1: $(pk, sk) \leftarrow\!\!\$\ \mathsf{Gen}()$
2: $(m_0, m_1) \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}(sk,\cdot)}(pk)$
3: $b \leftarrow \{0, 1\}$
4: $c_b \leftarrow \mathsf{Enc}(pk, m_b)$
5: $b' \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}(sk,\cdot)}(pk, c_b)$
6: **return** $b' = b$

$\underline{\mathcal{O}(sk, c)}$

1: **if** $c \neq c_b$ **then**
2: | **return** $\mathsf{Dec}(sk, c)$
3: **else**
4: | **return** $\perp$

IND-CCA is the strongest of the three security notions. It involves an adversary with the ability to encrypt arbitrary messages and with access to a decryption oracle, $\mathcal{O}$. The adversary can request the decryption of arbitrary messages, with exception of the challenge ciphertext. A scheme is IND-CCA-secure if the adversary is unable to distinguish which of two adversary-chosen plaintexts corresponds to the challenge ciphertext.

The schemes described in Sections 2.1.3.1 and 2.1.4 are CPA-secure only and, thus insecure against active adversaries. In the majority of applications where key exchanges are concerned, as is the case for Kyber and Swoosh, IND-CCA security is preferred or even required. This is due to the fact that adversaries with only passive capabilities do not accurately represent real-world scenarios.

[5]The adaptive CCA version is also commonly referred to as IND-CCA2.

In the case of KYBER, the CPA-secure PKE scheme is converted to a CCA-secure KEM through the application of the Fujisaki-Okamoto (FO) transform [100, 101]. The main idea behind this transform is to anchor the randomness employed by the encryption algorithm to the plaintext being encrypted, thus allowing the decapsulation algorithm to distinguish between honestly and dishonestly generated ciphertexts.

2.2 KEM vs. NIKE

Public-key cryptography, also known as asymmetric cryptography, allows two or more parties to communicate securely using a combination of public and private keys: the former used to encrypt data and the latter used to decrypt it. In addition, digital signature schemes can be used to attain further desirable properties, such as authentication, integrity and non-deniability. In some cases, public-key cryptography schemes are impractical and secret-key cryptography schemes are preferred. However, the latter requires both parties to agree on a shared secret key before they can communicate securely. This can be accomplished through key establishment mechanisms, including Key-Encapsulation Mechanisms (KEMs) and Non-Interactive Key Exchange (NIKE) schemes. A significant distinction between these two primitives is whether the parties can compute the shared secret independently once public keys are available. In a NIKE scheme, each party computes the shared secret using its own secret key and the other party's public key. In contrast, a KEM allows one party to generate a fresh shared secret and an associated encapsulation ciphertext using the recipient's public key; the recipient obtains the same shared secret by decapsulating the ciphertext with its secret key. Thus, KEM-based key establishment requires the transmission of a per-session encapsulation.

Definition 11 (Non-Interactive Key Exchange). *A NIKE scheme is defined as a pair of algorithms* $(\mathsf{KeyGen}, \mathsf{SharedKeyDeriv})$ *with the following definitions:*

$(sk, pk) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$**:** *key generation, a probabilistic algorithm that, given a security parameter* 1^λ*, computes a key pair composed of a secret key and a public key (sk and pk respectively),*

$k \leftarrow \mathsf{SharedKeyDeriv}(pk_1, sk_2)$**:** *shared key derivation, a deterministic algorithm that, given a public key* pk_1 *and a secret key* sk_2*, computes a shared key* k*.*

Definition 12 (NIKE Correctness). *A NIKE scheme is correct if, given two key pairs* (sk_1, pk_1) *and* (sk_2, pk_2)*, the shared-key derivation function outputs the same shared key for both parties, with overwhelming probability:* $\Pr[\mathsf{SharedKeyDeriv}(pk_1, sk_2) = \mathsf{SharedKeyDeriv}(pk_2, sk_1) : (sk_2, pk_2) \leftarrow \mathsf{KeyGen}(1^\lambda); (sk_1, pk_1) \leftarrow \mathsf{KeyGen}(1^\lambda)]$.

Definition 13 (Key-Encapsulation Mechanism). *A KEM scheme is defined as an algorithm triple* $(\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$ *with the following definitions:*

$(sk, pk) \leftarrow\!\!\$\ \mathsf{KeyGen}(1^\lambda)$**:** *key generation, a probabilistic algorithm that, given a security parameter* 1^λ*, computes a key pair composed of a secret key and a public key (sk and pk respectively).*

$(ct, k) \leftarrow\!\!\$\ \mathsf{Encaps}(pk)$**:** *encapsulation, a probabilistic algorithm that, given a public key pk, computes a ciphertext and a shared key (ct and k respectively).*

$k \leftarrow \mathsf{Decaps}(ct, sk)$**:** *decapsulation, a deterministic algorithm that, given a ciphertext ct and a private key sk, computes a shared key k.*

Definition 14 (KEM Correctness). *A KEM is said to be correct if, for a given key pair, the shared key computed by the decapsulation function is the same as the one computed by the encapsulation function, when the former is given the ciphertext of the latter as input, with overwhelming probability:* $\Pr[k = k' : k' \leftarrow \mathsf{Decaps}(ct, sk); (ct, k) \leftarrow\!\!\$\ \mathsf{Encaps}(pk); (sk, pk) \leftarrow\!\!\$\ \mathsf{KeyGen}(1^\lambda)]$.

These definitions highlight the main distinction between NIKEs and KEMs. NIKEs derive the shared secret directly from the parties' key pairs, while KEMs establish a fresh key through an encapsulation ciphertext, which must be transmitted to the recipient. Using the results in [98], one can convert a NIKE into an IND-CCA-secure KEM.

2.3 High-Assurance Implementations

Cryptographic mechanisms, whether protocols or primitives, are high-risk targets to attackers looking to exploit design flaws, bugs in implementations, and side-channel vulnerabilities. The lifecycle of such a mechanism, from the design to the deployment phase, presents a lot of opportunities for the introduction of errors which can lead to security issues.

1. **Design**: In the initial phase, when the protocol or primitive is being designed, erroneous assumptions about the security properties or mistakes in traditional pen-and-paper proofs can lead to the design of insecure protocols or primitives.

2. **Implementation**: Starting from a specification that has been proven secure can still lead to insecure implementations when these do not replicate the specification correctly. In the case of cryptographic primitives, reference implementations are less prone to deviate from the original specification as they tend to be one-to-one translations. On the other hand, optimized implementations, which are commonly used in applications,

introduce complexity which makes them harder to verify using traditional techniques, such as code auditing and software testing, and are more likely to introduce errors. Additionally, assumptions that are made at the specification and design level are often disconnected from the implementation, which can lead to further issues.

3. **Deployment**: Lastly, even when protocols and primitives have been proven secure and their respective implementations have been correctly implemented, certain properties of these applications, such as the timing of specific operations or memory access patterns, may reveal information that leads to vulnerabilities, through so-called side-channel attacks.

It is evident that these three phases present possible attack vectors that should be considered. Conventionally, these issues have been addressed through a combination of pen-and-paper proofs for analyzing the design security of the mechanisms at a high level, and software testing aimed at verifying the correctness and security properties of implementations. However, pen-and-paper proofs are susceptible to errors and are often based on abstract versions of the systems being analyzed, with limited connection to concrete implementations. Additionally, the methods employed for testing the correctness and security properties of these implementations may lack the capacity to identify edge cases.

Due to the critical nature of cryptographic mechanisms, it is imperative to enhance the current methodology to achieve higher assurance implementations.

2.3.1 Methodologies

One potential solution to these challenges can be found in the domain of computer-aided cryptography (CAC). This field utilizes formal methods to achieve a heightened degree of confidence in the properties previously delineated. At the design level, computer-verified security proofs can help address the complexity of the systems and prevent flaws commonly found in pen-and-paper proofs. At the implementation level, functional-correctness proofs provide assurance with respect to the equivalence of aggressively optimized code with regard to the original specification. Finally, memory safety and cryptographic constant-time guarantees, as well as protection against other side-channel vulnerabilities in deployed applications, can be attained through a combination of analysis and verification tools, as well as language features to enforce these properties.

The combination of these three pillars, as delineated in [35], constitutes the “Holy Grail” or “Grand Slam” of computer-aided cryptography. This term refers to a holistic approach to security analysis that encompasses design-level security, functional correctness, efficiency, and side-channel resistance for a particular implementation. The following sections will focus on the design and implementation levels, with an emphasis on the latter.

The employment of formal methods to achieve high-assurance cryptography can be categorized into several distinct approaches, each of which possesses its own set of advantages and disadvantages, as well as distinct objectives. Some of these tools are designed with the purpose of providing assurance with respect to the design security, such as Tamarin [188, 160], and ProVerif [58]. Others are intended to ensure the correctness and security of implementations namely CryptoLine [199, 99], and Hax [55]. Finally, there are tools that address more than one of the aforementioned pillars, such as EasyCrypt [37], and F* [196].

2.3.1.1 Security Proofs

The security analysis of cryptographic mechanisms using formal methods aims to ensure these constructions satisfy their stated security goals against a select group of attackers, by accurately modelling both the protocol and the attacker's capabilities. It should be noted, however, that the scope of the guarantees provided by CAC is limited to the statements being proved. If the security notion, model, or theorem statement does not functionally capture the intended property, then the proof assistant may verify a statement that is weatker than the desired security guarantee.

In [34, 31], the authors used the EasyCrypt proof assistant to reason about the security of XMSS [123] and SPHINCS$^+$ [50] respectively, after a flaw had been found in the original security proofs [138]. In the case of [34], the authors also constructed a new tight security proof for XMSS, which had not been provided in [121], where the aforementioned flaw was fixed. These works illustrate both the value and the scope of machine-checked security proofs: an invalid proof step would likely be rejected by the proof assistant, but the resulting guarantee remains tied to the correctness of the formalized model, assumptions, and security statement. For example, if one intended to prove adaptive IND-CCA security for a KEM such as Kyber, but formalized a game in which the adversary only receives decapsulation access before the challenge ciphertext, then the resulting proof would only establish non-adaptive IND-CCA security, instead of the intended adaptive IND-CCA. The proof assistant may accept the proof, but the verified statement would not capture the desired security property.

2.3.1.2 Correctness Proofs

A primary benefit of conducting a security analysis with CAC tools is the capacity to establish a connection with implementation-related proofs. Although pen-and-paper proofs are effective for reasoning about the high-level mathematical construction of a scheme, they typically operate at a level of abstraction that obscures implementation details. Operations such as field arithmetic, serialization, modular reduction, sampling, and bit manipulation are usually treated as idealized operations and are presumed to be correct. This abstraction, while useful for assessing the security of a scheme, limits the

ability to reason about the correctness of concrete implementations of that scheme. Errors in these implementations can introduce mismatches between the idealized construction that was proved secure and the program that is actually executed. Consequently, a security proof may not be applicable to the flawed implementation. CAC tools, such as EasyCrypt, help address this gap by enabling reasoning about both the security of a construction and the connection between it and a concrete instantiation, implementation, or specification. This capacity to reason about the correctness of an implementation constitutes a primary benefit of these tools. As previously stated, this is of particular importance for optimized implementations, which often deviate from the scheme's high-level specification in the manner in which operations are performed. Consequently, these implementations can prove more challenging to verify. Furthermore, linking specification and implementation can serve to eliminate subtle discrepancies, even in reference implementations, such as the one observed in many implementations of the SABER scheme [108, Sec. 5.4]. As previously discussed in [35], testing and code auditing methodologies, while useful, tend to be unable to provide sufficient coverage to detect corner cases, such as a bug in the PQM4 [131] implementation of Kyber1024 that led to an overflow not triggered by any test vectors [130]. It is imperative to acknowledge that, despite its efficacy as a strategy for attaining higher assurance, program verification is not without its limitations and potential pitfalls. These limitations include improper modeling of implementations, discrepancies between the model and the source code (and/or machine code), or the presence of bugs in the verification tools or other parts of the trusted computing base (TCB) themselves.

Approaches to program verification of correctness can be categorized into two distinct modes. The first is functional correctness, where an implementation is proved to satisfy an abstract mathematical specification of the intended algorithm or primitive. In this setting, the specification is written in a high-level mathematical form, and the goal is to show that, for all valid inputs, the program being verified computes the function being described. The second approach is equivalence checking, where the goal is to prove two programs behave identically. In this settting, one of the programs is usually a reference implementation or imperative specification that has been proven correct or is implicitly trusted. The selection of which method to use is typically contingent upon the source code being verified. Reference implementations, which are written to closely resemble the specification without significant deviation, are typically verified via functional correctness, as the proof effort is relatively straightforward. Conversely, efficient implementations that utilize aggressive optimizations are typically best verified through equivalence checking with the reference implementation. Depending on the verification effort, this process generally entails the development of intermediate representations of the optimized code, each one progressively closer, syntactically, to the reference implementation or model of the scheme. These models are linked through the

use of equivalence proofs, a process that ultimately results in the generation of a proof by transitivity, as illustrated in figure Figure 2.1

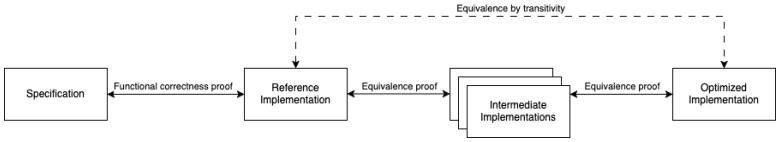


Figure 2.1: Correctness proofs.

Furthermore, it is important to note that, in many cases, the functional correctness of the reference implementation is also typically validated through a series of hops between equivalent specifications of the target scheme at varying levels of abstraction.

2.3.2 Formosa Toolchain

The Formosa toolchain proposes a comprehensive approach to high-assurance cryptography, integrating a formally verified compiler and assembly-like language, Jasmin [15], to develop efficient implementations, with the EasyCrypt [37] toolset, to verify their correctness as well as other security properties, such as cryptographic constant time. One of the key features of this toolchain, and specifically of the Jasmin framework, is the ability to extract programs to EasyCrypt in order to reason about the cryptographic systems being implemented, enabling proofs of correctness as well as linking these to security proofs. This extraction, although not verified, is a one-to-one translation of the Jasmin program to an EasyCrypt module. The Sections 2.3.2.1 and 2.3.2.2 introduce Jasmin and EasyCrypt respectively, using an implementation of the One-Time Pad (OTP) cipher as a running example. This cipher can be used to encrypt an arbitrarily long message using a pre-shared secret key. The key, called the one-time pad, must be randomly generated and at least as long as the plain text, is XOR'ed together with the message to compute the ciphertext, as shown in Algorithm 6.

Algorithm 6 One-Time Pad

Require: Key $k \in \{0,1\}^l$, Message $m \in \{0,1\}^l$
Ensure: Ciphertext $c \in \{0,1\}^l$
1: **for** $i = 0 \rightarrow l - 1$ **do**
2: $\quad c_i \leftarrow k_i \oplus m_i$
3: **return** c

2.3.2.1 Jasmin

The Jasmin framework includes a programming language, a compiler, and a collection of tools designed for the development of high-assurance and efficient cryptographic primitives. The compiler provides support for AMD64, ARMv7, and RISC-V targets, along with Instruction-Set-Architecture (ISA) extensions such as AVX2 for the AMD64 architecture, and is formally verified to preserve the semantics of the source program. The Jasmin compiler and type system ensure that cryptographic constant-time code, at the source level, remains constant time at the target/assembly level. These properties are also formally proven to be preserved by the compiler [193]. In order to facilitate the development of efficient implementations, the Jasmin language incorporates a set of low-level features that provide fine-grained control over register allocation and instruction selection. These features are complemented by abstractions such as functions, arrays, sub-arrays, and loops, which are common in higher-level languages and facilitate development. However, in contrast to other languages, the majority of these abstractions incur no performance penalties, as the compiler includes a set of passes that unroll or inline loops and function calls. This, in conjunction with the aforementioned low-level features, means the assembly code generated by the Jasmin compiler is close to a direct line-to-line translation of the source code. It should be noted that a limitation of this emphasis on low-level features is that, in general, Jasmin implementations are architecture-specific. Consequently, unlike higher-level languages, they cannot be readily compiled to a different target architecture without modifications to the source code.

Figure 2.2 illustrates a Jasmin implementation of the One-Time Pad cipher. In essence, the implementation of this cipher encrypts the input message in 32-byte increments, as set by the BLOCK_SIZE constant. In the event that there are fewer than 32 bytes remaining, the implementation encrypts each of the remaining bytes individually. The ensuing paragraphs delineate the previously mentioned implementation and introduce the Jasmin language.

```
1   param int BLOCK_SIZE = 32;
2   param int LEN = 300;
3
4   inline fn __enc_block(reg ptr u8[BLOCK_SIZE] ct msg sk) -> reg ptr u8[BLOCK_SIZE] {
5     inline int i;
6     reg u8 c;
7
8     for i=0 to BLOCK_SIZE {
9       c = msg[i];
10      c ^= sk[i];
11      ct[i] = c;
12    }
13
14    return ct;
15  }
16
17  fn _enc_bytes(reg ptr u8[LEN] ct msg sk, reg u64 i) -> reg ptr u8[LEN] {
18    reg u8 c;
19
20    while(i < LEN) {
21      c = msg[i];
22      c ^= sk[i];
23      ct[i] = c;
24      i += 1;
25    }
26
27    return ct;
28  }
29
30  export fn otp(reg ptr u8[LEN] ct msg sk) -> reg ptr u8[LEN] {
31    inline int i;
32    reg u64 j;
33
34    for i=0 to LEN/BLOCK_SIZE {
35      ct[i*BLOCK_SIZE:BLOCK_SIZE] = __enc_block(ct[i*BLOCK_SIZE:BLOCK_SIZE],
36                                                msg[i*BLOCK_SIZE:BLOCK_SIZE],
37                                                sk[i*BLOCK_SIZE:BLOCK_SIZE]);
38    }
39
40    j = LEN/BLOCK_SIZE*BLOCK_SIZE;
41    ct = _enc_bytes(ct, msg, sk, j);
42    return ct;
43  }
```

Figure 2.2: One-Time Pad Jasmin Reference Implementation

Functions in Jasmin are defined using a syntax similar to that of Rust. The **fn** keyword is followed by the function name, the parameters in parentheses and finally, the output value types are indicated by the -> string. The body of the function is delimited by curly brackets thereafter. Jasmin provides three distinct function types: export functions, inline functions and local functions, which are illustrated in the block. Parameter and variable declaration in Jasmin starts by stipulating the desired allocation for the variable, reg or stack, Functions of type export, such as otp (Figure 2.2, line 30), are entry-points that can be invoked from other programs. These functions must follow a specific ABI (Application Binary Interface), thereby limiting the number of arguments and return values. The types of arguments to export functions are limited to machine words (e.g. u64, u32, u16, u8, etc) and pointers which are indicated by the type ptr. These

value types are used to pass information between Jasmin and the programs that invoke it, in the form of pointers or integers. In essence, export functions commonly serve as a thin wrapper, loading data pointed to by its parameters into Jasmin's internal memory, such as stack arrays. Subsequently, these functions invoke one of the local or inline function types to perform the computation. The result can be stored in a memory location specified as input, such as the memory location designated as ct in the OTP example. Inline functions, such as __enc_block (Figure 2.2, line 4), are, as the name implies, inlined at compile time and thus generate no function call. Consequently, these calls exhibit no run-time overhead as their bodies are inlined at every call site. On the other hand, local functions such as _enc_bytes (Figure 2.2, line 17) are preserved during the compilation process without being exposed to external programs. As a consequence, these functions are not subject to limitations regarding the number of arguments and return values, as far as ABI conventions go.

Parameter and variable declaration in Jasmin starts by stipulating the desired allocation for the variable, reg or stack, which indicate whether the variable ought to be stored in a register or stack respectively. Subsequently, the data type of the variable is specified, along with its name. For instance, reg u64 j (Figure 2.2, line 32) declares a variable named j, of type u64 that ought to be stored in a register. As previously stated, in Jasmin it is the responsibility of the programmer to ensure that there are sufficient registers available for the variables that ought to be stored in one, and to spill variables to memory when this is not the case. Consequently, in instances where the number of live register variables exceeds the number of available registers, compilation will fail, and variables must be spilled to the stack. In addition to variables, Jasmin offers a specialized value type denoted as inline int. This type, as the name implies, indicates integer values that are inlined at compile time. These values are primarily utilized within the context of **for** loops, which are themselves also unrolled at compile time and thus generate no branching instructions.

As illustrated in Figure 2.2, the function __enc_block utilizes a **for** loop (Figure 2.2, line 8) to sequentially load a single byte of the message into register c, XOR it with the secret key sk, storing the resulting ciphertext in ct. Once there are fewer than BLOCK_SIZE bytes remaining, the otp function invokes the __enc_bytes function to encrypt the remaining bytes. This function works similarly to the __enc_block function, but it encrypts only the remaining bytes. Consequently, in constrast to a **for** loop, which requires a known number of iterations in order to be unrolled at compile time, a **while** loop (Figure 2.2, line 20) is used to encrypt the remaining bytes, denoted by LEN.

A critical consideration in the implementation of cryptographic primitives pertains to the performance of these implementations. There are several strategies and levels at which optimizations occur, from developing more efficient techniques or algorithms to perform the same functionality, to the direct selection of instructions that execute operations more efficiently or

on more data simultaneously. The latter, also referred to as vectorization, typically involves the use of SIMD (Single Instruction Multiple Data) extensions available in modern CPUs. One such extension, common in AMD64, is AVX2 (Advanced Vector Extensions 2), which allows processing multiple data elements simultaneously using vector registers which contain elements of smaller sizes. For example, a 256-bit register has the capacity to store 16 16-bit values on which vector instructions operate, simultaneously. For instance, VPADD enables the addition of several elements of a predefined size in parallel. Jasmin gives the programmer direct access to these instructions. To illustrate this point, Figure 2.3 provides a vectorized implementation of the __enc_block function in Figure 2.2. The key difference between this implementation and the one in Figure 2.2 is the replacement of the scalar XOR instruction (Figure 2.2, line 11) with its vector counterpart, VPXOR_256 (Figure 2.3, line 8), which performs the XOR operation on 32-bytes simultaneously. Consequently, the **for** loop in Figure 2.3 executes only BLOCK_SIZE/32 iterations, as opposed to BLOCK_SIZE.

```
1   inline fn __enc_block(reg ptr u8[BLOCK_SIZE] ct msg sk) -> reg ptr u8[BLOCK_SIZE] {
2     reg u256 a b c;
3     inline int i;
4
5     for i=0 to BLOCK_SIZE/32 {
6       a = msg[u256 i];
7       b = sk[u256 i];
8       c = #VPXOR_256(a, b);
9       ct[u256 i]  = c;
10    }
11
12    return ct;
13  }
```

Figure 2.3: One-Time Pad Jasmin Vectorized Implementation

Although the optimization in question is relatively straightforward, the integration of vector instructions within more complex cryptographic schemes frequently results in implementations that deviate significantly from the reference implementation and/or the specification of the scheme. This discrepancy can potentially lead to errors in the optimized code, resulting in an inaccurate implementation of the target scheme. As shown in Figure 2.1 and Section 2.3.2, the use of tools such as EasyCrypt to write proofs of functional correctness can help overcome this obstacle by checking the equivalence of these optimized implementations with reference ones, that have been proven correct.

2.3.2.2 EasyCrypt

EasyCrypt is a proof assistant designed to facilitate the development of correctness and security proofs for cryptographic implementations and primitives. As mentioned previously, implementations written in Jasmin can

be automatically extracted to EasyCrypt models to allow reasoning about functional correctness and constant-time properties. Using these correctness proofs, the implementations can be linked to a specification which, once proven secure using EasyCrypt, coupled with the formal guarantees of semantic preservation, allows developers to establish an end-to-end proof of security with respect to the assembly code produced by the Jasmin compiler.

To illustrate this with the One-Time Pad example, Figure 2.4 presents the extracted EasyCrypt model of the reference implementation of Figure 2.2.

```
1   proc __enc_block (ct:BArray32.t, msg:BArray32.t, sk:BArray32.t) : BArray32.t = {
2     var i:int;
3     var c:W8.t;
4     i <- 0;
5     while ((i < 32)) {
6       c <- (BArray32.get8 msg i);
7       c <- (c `^` (BArray32.get8 sk i));
8       ct <- (BArray32.set8 ct i c);
9       i <- (i + 1);
10    }
11    return ct;
12  }
13  proc _enc_bytes (ct:BArray300.t, msg:BArray300.t, sk:BArray300.t, i:W64.t) :
14  BArray300.t = {
15    var c:W8.t;
16    while ((i \ult (W64.of_int 300))) {
17      c <- (BArray300.get8 msg (W64.to_uint i));
18      c <- (c `^` (BArray300.get8 sk (W64.to_uint i)));
19      ct <- (BArray300.set8 ct (W64.to_uint i) c);
20      i <- (i + (W64.of_int 1));
21    }
22    return ct;
23  }
24  proc otp (ct:BArray300.t, msg:BArray300.t, sk:BArray300.t) : BArray300.t = {
25    var aux:BArray32.t;
26    var inc:int;
27    var i:int;
28    var j:W64.t;
29    inc <- (300 %/ 32);
30    i <- 0;
31    while ((i < inc)) {
32      aux <@ __enc_block ((SBArray300_32.get_sub8 ct (i * 32)),
33      (SBArray300_32.get_sub8 msg (i * 32)),
34      (SBArray300_32.get_sub8 sk (i * 32)));
35      ct <- (SBArray300_32.set_sub8 ct (i * 32) aux);
36      i <- (i + 1);
37    }
38    j <- (W64.of_int ((300 %/ 32) * 32));
39    ct <@ _enc_bytes (ct, msg, sk, j);
40    return ct;
41  }
```

Figure 2.4: EasyCrypt model of reference One-Time Pad

This extraction is automatically generated by the Jasmin compiler, requiring no manual effort on the developer side. Although not verified, the extraction is a one-to-one translation of the source Jasmin program.

In order to reason about the properties of such a model, EasyCrypt utilizes three different variants of Hoare logic:

- HL, deterministic Hoare logic, to reason about properties of deterministic programs using statements of the form **hoare** [M.foo : P ==> Q], which states that for every initial state satisfying predicate P, after the execution of M.foo, the predicate Q is satisfied, if M.foo terminates.

- pHL, probabilistic Hoare logic, to reason about programs with probabilistic behavior, using statements of the form **phoare** [M.foo: P ==> Q] ⋄ p, where ⋄ is a comparison operator (such as < and =) and p an expression representing a probability. Here the judgement indicates that for every initial state satisfying predicate P, the probability that the execution of M.foo terminates in a state that satisfies the predicate Q is related to p by the operator ⋄.

- pRHL, probabilistic relational Hoare logic, to reason about pairs of programs with probabilistic behavior, M.foo and N.foo, using statements of the form **equiv** [M.foo ~ N.foo : P ==> Q]. A judgement of such a form states that, for every pair of initial states satisfying the predicate P, the output distributions produced by the executions of M.foo and N.foo, can be related such that the predicate Q is satisfied.

In EasyCrypt, statements used to prove properties concerning a program, or a pair of programs, are designated as lemmas. The establishment of functional-correctness proofs typically involves the utilization of both HL and pHL lemmas, with the former being used to prove a certain property about the program and the latter used to prove that the program terminates with probability 1. Conversely, correctness proofs via equivalence are attained utilizing pRHL and pHL lemmas: program P_2 is correct if it is equivalent to program P_1 and P_1 has been proven correct. In essence, this equates to the strategy delineated in Figure 2.1.

The process of writing proofs in EasyCrypt involves the manipulation of the proof state until the post condition is satisfied. This process entails transitioning between analogous states by employing a series of tactics and lemmas.

To illustrate how to accomplish this in the One-Time Pad example, P_1 now refers to the reference implementation (Figure 2.2) and P_2 refers to the vectorized implementation (Figure 2.3) of this scheme. The process of verifying correctness for both implementations can be divided in four steps:

1. Prove the reference implementation of the OTP (line 30 in Figure 2.2) is functionally correct w.r.t (satisfies) the OTP specification.

2. Prove the vectorized implementation of __enc_block is equivalent to the reference implementation.

3. Prove the vectorized implementation of OTP (that uses __enc_block) is equivalent to the reference implementation.

4. Derive that the vectorized implementation is functionally correct by transitivity.

The initial step of in this procedure is to formulate an EasyCrypt specification for the One-Time Pad, as illustrated in Figure 2.5.

```
1   type plaintext = BArray300.t.
2   type skey = BArray300.t.
3   type ciphertext = BArray300.t.
4
5   op otpb : W8.t -> W8.t -> W8.t = W8.(`^`)
6   axiomatized by otpbE.
7
8   op OTP (msg: plaintext) (sk: skey) = BArray300.map2 otpb msg sk
9   axiomatized by OTPE.
```

Figure 2.5: Functional EasyCrypt specification of OTP

For the sake of simplicity, this specification assumes that messages, keys and ciphertexts are 300 bytes in length (lines 1-3). The OTP operator, in line 8, defines the scheme as a mapping from the message and secret-key domains to the ciphertext domain using the XOR operation to combine the two inputs, one byte at a time (line 5). This specification is functional in nature, as it delineates the desired result as a function of its inputs. While this is a useful and natural way of describing the One-Time Pad, implementations in an imperative language such as Jasmin deviate from this specification significantly. As a consequence, it is advantageous to propose an alternative, but equivalent, specification that more closely resembles the syntax of these implementations. Figure 2.6 presents an imperative specification of the One-Time Pad which, once proven to correctly implement the OTP operator in Figure 2.5, can be used as a reference to reason about Jasmin implementations.

```
1   proc otp (msg: plaintext, sk: skey) = {
2     var i: int;
3     var ctxt: ciphertext;
4
5     i <- 0;
6
7     while(i < 300) {
8       ctxt <- BArray300.set8 ctxt i (msg[i] `^` sk[i]);
9       i <- i + 1;
10    }
11
12    return ctxt;
13  }
```

Figure 2.6: Imperative EasyCrypt specification of OTP

A comparison of the specified model with the EasyCrypt model extracted from the reference implementation reveals a high degree of similarity, thereby facilitating the equivalence proof. This equivalence is captured by the otp_ref_corr lemma in Figure 2.7 (line 1), which postulates that, given the same message msg and secret key sk, both the imperative specification (Otp.otp) and the reference implementation (Jotp_ref.M.otp) of the OTP produce the same result (ciphertext).

```
equiv otp_ref_corr:
  Jotp_ref.M.otp ~ Otp.otp:
    ={msg, sk} ==> ={res}.
```

Figure 2.7: Correctness lemma of reference OTP implementation

Proving such a lemma, when dealing with two programs that utilize loops, generally entails reasoning about the loop invariant of each program, such that once both loops have terminated, the result of both programs is equivalent. This approach is analogous to the strategy employed to prove the equivalence between the reference and the vectorized implementations, after establishing the equivalence of the __enc_block function. The EasyCrypt model of the vectorized implementation of this function is shown in Figure 2.8. The remaining code is omitted as it is identical in both implementations.

```
1   proc __enc_block (ct:BArray32.t, msg:BArray32.t, sk:BArray32.t) : BArray32.t = {
2     var inc:int;
3     var i:int;
4     var a:W256.t;
5     var b:W256.t;
6     var c:W256.t;
7     inc <- (32 %/ 32);
8     i <- 0;
9     while ((i < inc)) {
10      a <- (BArray32.get256 msg i);
11      b <- (BArray32.get256 sk i);
12      c <- (VPXOR_256 a b);
13      ct <- (BArray32.set256 ct i c);
14      i <- (i + 1);
15    }
16    return ct;
17  }
```

Figure 2.8: EasyCrypt model of vectorized One-Time Pad

The equivalence proof between the vectorized and the reference implementation of the __enc_block function is depicted in Figure 2.9.

```
1   equiv eq_enc_block :
2     Jotp_vec.M.__enc_block ~ Jotp_ref.M.__enc_block :
3       ={msg,sk} ==> ={res}.
4   proof.
5     proc.
6     cfold {1} 1.
7     unroll for{2} 2.
8     unroll for{1} 2.
9     auto => /> *.
10    apply all_eqP; rewrite /all_eq /=.
11    by do 32 !(rewrite get8_set256dE /= get256d_byte 1://= get256d_byte 1://= /=).
12  qed.
```

Figure 2.9: Equivalence proof of __enc_block function

Given that the number of iterations of the loop used in these functions is known at compile time, the strategy employed in this proof entails unrolling both loops and comparing the resulting bytes, one by one. A critical aspect to consider when undertaking correctness proofs of Jasmin implementations is the specification of the ISA in EasyCrypt. This specification, which is part of the TCB, utilizes EasyCrypt operators to represent instructions, which are replaced upon extraction to EasyCrypt. The equivalence result of eq_enc_block is utilized as a building block in the proof that demonstrates the vectorized implementation of the OTP is equivalent to the reference implementation. The lemma in Figure 2.10 is employed to express this equivalence by stating that, given the same input parameters (message and secret key) ={msg,sk}, both programs output the same result (ciphertext) ={res}.

```
equiv otp_vec_eq:
  Jotp_vec.M.otp ~ Jotp_ref.M.otp:
    ={msg, sk} ==> ={res}.
```

Figure 2.10: Equivalence lemma of OTP in EasyCrypt

After demonstrating the equivalence between the vectorized and reference implementation, the functional correctness of the former is derived via the transitive property. In a manner analogous to the lemma in Figure 2.7, the following lemma stipulates that the vectorized implementation is correct if, given the same inputs, it yields the same result as the specification. This proof is predicated on the equivalence proof in Figure 2.10, which is used to substitute the vectorized implementation with the reference implementation (line 9). Subsequently, the correctness proof of the reference implementation (Figure 2.7) is used to conclude the lemma (line 10).

```
1   equiv otp_vec_corr:
2     Jotp_vec.M.otp ~ Otp.otp:
3       ={msg, sk} ==> ={res}.
4   proof.
5     proc * => /=.
6     transitivity {1} { r <@ Jotp_ref.M.otp(ct, msg , sk); }
7       (={msg, sk} ==> ={r})
8       (={msg, sk} ==> ={r}); 1,2: by smt().
9     + by call otp_vec_eq.
10    + by call otp_ref_corr.
11  qed.
```

Figure 2.11: Correctness lemma of vectorized OTP implementation

Therefore, it can be inferred that the vectorized implementation of the OTP in Figure 2.8 is equivalent to the specification of the scheme in Figure 2.6 and, consequently, correct.

Although the main focus of this thesis is implementation correctness, EasyCrypt can also be used prove design-security properties, and thus achieve the "Holy Grail" of computer-aided cryptography. In the example above, the EasyCrypt specifications in Figures 2.5 and 2.6 are treated as faithful representations of the OTP described in Algorithm 6, and the proof establishes that the Jasmin implementation is equivalent to these specifications. This establishes functional correctness, but it does not by itself show that the specified scheme is secure. However, as mentioned previously, such assumptions can sometimes be erroneous and even when the scheme being described is proven secure using pen-and-paper methods, its translation to EasyCrypt can introduce mistakes or deviations that invalidate the intended security argument.

In EasyCrypt, one could formalize an appropriate security game for the OTP, such as perfect secrecy or one-time indistinguishability under the assumption that the key is uniformly random, has the same length as the message, and is never reused. Proving that the EasyCrypt specification satisfies this security notion would establish the security of the abstract scheme in Figure 2.5. Combined with the equivalence proof between the EasyCrypt specification and the Jasmin implementation, this would yield an end-to-end argument: the specification is secure, the Jasmin implementation correctly realizes the specification, and therefore the implementation inherits the proved security property, up to the assumptions captured by the model.

3

Formally verifying Kyber – Episode IV: Implementation correctness

This chapter is for all practical purposes identical to the paper *"Formally verifying Kyber – Episode IV: Implementation correctness"* [129], co-authored with José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Published at TCHES 2023.

Abstract. In this paper we present the first formally verified implementations of Kyber and, to the best of our knowledge, the first such implementations of any post-quantum cryptosystem. We give a (readable) formal specification of Kyber in the EasyCrypt proof assistant, which is syntactically very close to the pseudocode description of the scheme as given in the most recent version of the NIST submission. We present high-assurance open-source implementations of Kyber written in the Jasmin language, along with machine-checked proofs that they are functionally correct with respect to the EasyCrypt specification.

We describe a number of improvements to the EasyCrypt and Jasmin frameworks that were needed for this implementation and verification effort, and we present detailed benchmarks of our implementations, showing that our code achieves performance close to existing hand-optimized implementations in C and assembly.

Keywords: High-assurance cryptography, lattice-based KEMs, NIST PQC, Jasmin, EasyCrypt

3.1 Introduction

In July 2022, NIST announced the first batch of "winners" of the post-quantum project, i.e., schemes that will be forwarded to standardization [3]. To the surprise of many, this first batch contained three signature schemes (CRYSTALS-Dilithium [41, 90], Falcon [97], and SPHINCS$^+$ [25, 50]),

but only one key-encapsulation mechanism (KEM): the lattice-based scheme CRYSTALS-Kyber (or in short just Kyber) [27, 65]. This is remarkable in particular because most experts see the biggest urgency in post-quantum deployment for KEMs rather than signatures. The reason for this urgency is the *store-now-decrypt-later* attack scenario that breaks the confidentiality of ciphertexts sent today when, some time in the future, attackers have a quantum computer in their arsenal. Consequently, most early-adopter experiments with post-quantum cryptography focus on key encapsulation [141, 142, 140, 171] and we also expect that the first big wave of post-quantum deployment in the next few years—certainly after the NIST standards are ready—will lead to widespread use of the one KEM covered by those standards: Kyber.

This deployment and the migration to post-quantum cryptography in general, presents a massive challenge for both technical and organizational reasons. Technical reasons include the fact that KEMs cannot in general be used as a drop-in replacement for ubiquitous Diffie-Hellman key exchange, the significantly larger sizes of public keys and ciphertexts compared to ECC, and the relative immaturity of side-channel protections. Examples of organizational reasons are modifications of protocol standards, updates to certification criteria, and strategies for updates of devices in the field.

However, this deployment of the next generation of asymmetric crypto also presents a huge opportunity to improve the deployed cryptographic software. Over the last decade, the field of *computer-aided cryptography* [35] has brought the use of formal methods into cryptographic engineering. This resulted in high-performance cryptographic implementations with strong formal guarantees of functional correctness and implementation-specific security properties such as memory safety or absence of timing leaks. Indeed, high-assurance crypto software is already being used in, for example, the NSS library used in the Firefox web browser [208, 51] and the BoringSSL library by Google [94]. While these examples show that high-assurance cryptography is ready for real-world deployment, a more widespread deployment requires replacing existing implementations. The costs of this shift are often hard to justify, despite the stronger security guarantees of formally verified code.

The situation is different for the deployment of post-quantum crypto, where new software *must* be rolled out. In this work we give an important step to enable the deployment of the next generation of cryptographic primitives relying also on a new generation of high-assurance post-quantum cryptographic software.

Contributions. In this paper we present the first fully formally verified implementations of Kyber and, to the best of our knowledge, the first such implementations of any post-quantum cryptosystem. Completing this work represented a significant challenge that implied non-trivial improvements to pre-existing formal verification tools. The challenges resided not only in the scale of the verified code — which, for example, makes it impossible to work with fully inlined functions — but also in a number of optimizations that do not appear

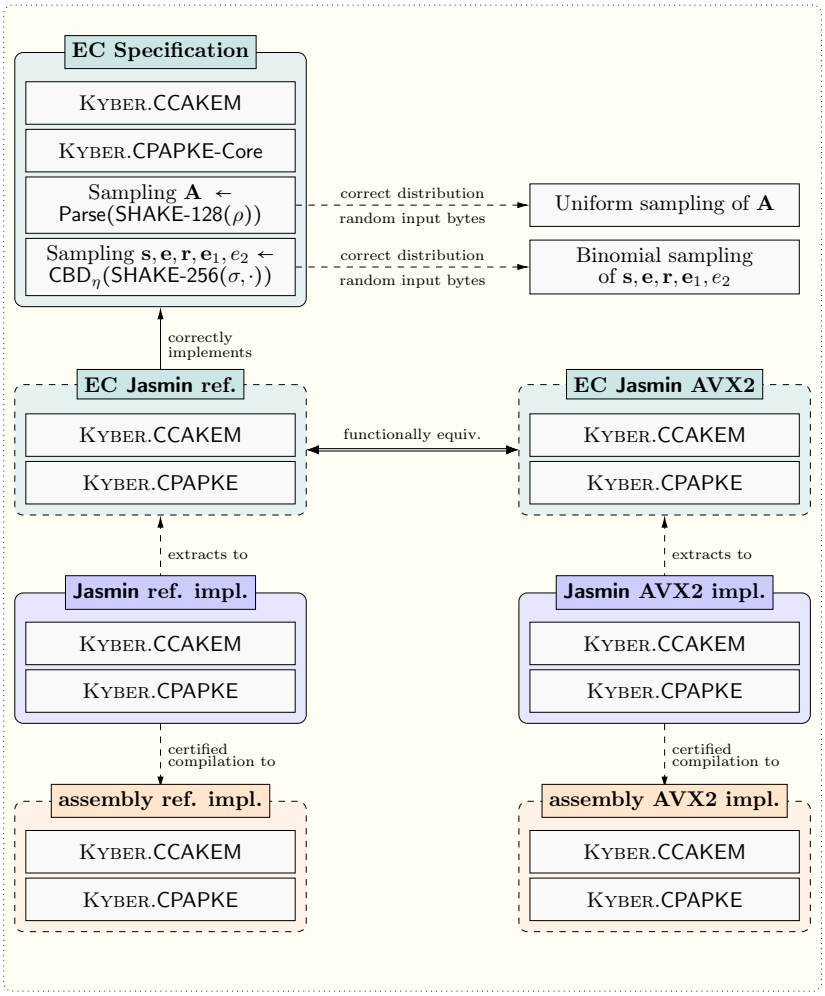


Figure 3.1: Overview of the proof structure and tooling. Elements in dashed boxes are automatically generated; elements in solid boxes need to be written by hand.

in pre-quantum cryptography, such as a fast Number Theoretic Transform (NTT), noise generation and rejection sampling, and data compression and decompression routines that have an impact in the correctness and security semantics.

More specifically, our contributions are the following:

1. We formalize the specification of KYBER [27, Sec. 1] in EASYCRYPT [37]. Great care (and effort) was dedicated to ensure that the EASYCRYPT

specification is syntactically close enough to the KYBER specification to enable efficient human validation. This means, in particular, that we use *algorithmic* definitions of the randomness sampling routines and provide computer-verified proofs that these algorithms generate outputs that follow the distributions defined in the KYBER specification.

2. We extend the Jasmin programming language [15, 18] with local functions, sub-arrays, pointer types required for call-by-reference, and system calls to obtain randomness. We communicated these extensions to the Jasmin developers and they have meanwhile been adopted upstream.
3. We present two implementations of KYBER768 in Jasmin targeting the AMD64 architecture: one reference implementation and one optimized implementation using the AVX2 vector instruction set. We use the automated extraction provided by the Jasmin compiler to obtain EASYCRYPT models of the two Jasmin implementations.
4. We extend EASYCRYPT's standard library with new features required by KYBER, such as support for polynomial rings, signed representations of field elements, dealing with sub-arrays, etc. We created a library that specifies and validates several optimizations of NTT computations used in KYBER, but which is of interest for other post-quantum cryptosystems. We also add better support for reasoning about nested for loops, and automatic inference of functional specifications for simple EASYCRYPT procedures. All of these enhancements reduce the need for human input when writing the proofs. We communicated these extensions to the EASYCRYPT developers and they have also been adopted upstream.
5. We then provide two computer-verified proofs to establish the functional correctness of our implementations: First we prove that the reference Jasmin implementation correctly implements the specification of KYBER in EASYCRYPT; Second we prove that the optimized AVX2 Jasmin implementation is equivalent to the reference implementation.
6. Finally we provide comparative benchmarks of our implementations on different generations of Intel CPUs. As we did not prove correctness of the rejection-sampling routine in the AVX2 implementation, we benchmark two variants of our AVX2 implementation: one that is fully optimized and one that is fully verified. (The latter uses rejection sampling from the reference implementation.) These benchmarks show that the fully optimized AVX2 Jasmin implementation is within about 10% of the performance of the hand-optimized C/asm implementation by the KYBER team.

An overview of our implementation and verification effort and the tooling we use is given in Figure 3.1.

What does our result mean? The use of the formally verified Jasmin compiler guarantees correct translation to assembly, so the behavior of the assembly

code is guaranteed to match the behavior of the source Jasmin code. The computer-verified proofs of functional correctness establish the link between the implementations and the specification of Kyber and, ultimately, guarantee that the assembly code is indeed performing the computations specified by the Kyber specification.

However, there are two aspects that still require careful manual inspection by expert auditors: the formal specification of Kyber in EasyCrypt and the statements of the theorems that we prove. We therefore dedicate most of the paper to explaining our formalization choices with respect to these two aspects, and present only a summarized view of the proof itself, focusing on the most innovative aspects and documenting lessons learned for the future.

With regards to verification of implementation security, we refer to [21], which reports formally proven protection against classical timing attacks and Spectre v1 attacks [133] for Jasmin implementations of multiple cryptographic routines, including Kyber. The implementations of Kyber considered in that paper are very close to the ones proven correct here. We are confident that we can use the same approach to prove constant-timeness and protect against Spectre v1, once the security type system in the Jasmin compiler from [21] is integrated into the main branch. Also, we are confident that the performance overhead for Spectre v1 protections will be just as low (i.e., $< 1\%$) as reported in [21].

Limitations and TCB. Our computer-verified proofs rely on a trusted code base (TCB) and are limited in scope to the Kyber specification. The correctness proofs of the Jasmin compiler rely on the assumption that the semantics of individual machine instructions are correctly formalized in Coq and EasyCrypt. The mechanism that extracts EasyCrypt code from Jasmin code is also part of the TCB. Finally, the EasyCrypt tool itself is part of the TCB. In terms of scope, our proofs do not cover the correctness of the SHA3 implementations, which come from the libjade library[6]. Some of this code has been verified in [17], but we do not establish a formal connection to this work.

We note also that the correctness theorem for the Jasmin compiler and the extraction to EasyCrypt only apply to memory-safe programs. However, automatically proving memory safety of our Kyber implementations is currently out of scope for the static safety analyzer in the Jasmin compiler. For this reason, we check memory safety using a combination of static analysis and dynamic analysis: the functional correctness proof guarantees that the memory accesses which are relevant for the computation of the scheme are safe, but they do not exclude accesses that cause unsafety due to, e.g., misalignment or simply accessing memory that is subsequently not used. For this reason we use the Jasmin interpreter and valgrind to reinforce the memory-safety guarantees.

The bigger picture. As the subtitle of this paper suggests, formally verifying the functional correctness and certain aspects of implementation security of

[6] `https://github.com/formosa-crypto/libjade`

Kyber is only part of the whole story. The use of EasyCrypt as specification language and verification tool allows us to also conduct mechanized reductionist proofs of security, i.e., establish through computer-verified proofs that Kyber is indeed an IND-CCA2 secure KEM. Our development is fully prepared so that, when the ongoing machine-checked proof of security is concluded, this yields a security theorem for the Kyber specification that, by the proofs presented in this paper, extends all the way down to the implementations at assembly level.

Related Work. While this paper is the first to present a fully computer-verified implementation of a post-quantum crypto scheme, it is not the first to present verification efforts in the realm of post-quantum cryptography. Hwang et al. [125] use CryptoLine[7] to formally verify the AVX2 implementation of the NTT transform used in the Kyber NIST submission. This is essentially the same code used for NTT in our AVX2 implementation, so our results corroborate the findings in [125]. However, our proof goes beyond NTT and covers two full Kyber implementations. We further discuss [125] in the concluding remarks of the paper. Kreuzer [137] focuses on machine-checking the correctness of the Kyber specification as a cryptographic construction, i.e., a proof that encryption and decryption commute. Other works have focused on machine-checking proofs of security for post-quantum cryptographic constructions, such as [201, 32, 122]. These works are orthogonal to what we present in this paper, since here we are focusing on the correctness of the Kyber implementations. A comprehensive description of related work in computer-aided cryptography can be found in [35], including prior uses of Jasmin to obtain verified high-speed implementations of symmetric and elliptic-curve cryptographic primitives, as well as alternative frameworks designed to achieve the same goal.

Artifact. An artifact including our extended versions of Jasmin and EasyCrypt, the Jasmin implementations of Kyber, and the EasyCrypt proofs is available from `https://artifacts.formosa-crypto.org`. We will also submit this artifact to the CHES artifact evaluation. The artifact includes a README with detailed instructions on how to build the tools, build the software, validate the proofs in EasyCrypt, and run our benchmarks. The contents of this artifact are under permissive licenses (MIT license for all modifications to EasyCrypt and CC0 for all Jasmin implementations and associated proofs).

3.2 Specification of Kyber in EasyCrypt

The EasyCrypt specification of Kyber closely follows the description in the latest version (3.02) of the CRYSTALS-Kyber proposal [27], omitting

[7] CryptoLine is a formal-verification framework that takes low-level code with user annotations for pre-conditions, post-conditions and intermediate assertions, and generates inputs for SMT solvers and computer algebra systems that can then be used to automatically check correctness. It is available from `https://github.com/fmlab-iis/cryptoline`.

```
module KyberKEM(HS : HSF.PseudoRF, XOF : XOF_t, PRF : PseudoRF,
    KemHS : HSF_KEM.PseudoRF, KemH : KEMHashes, O : RO.POracle)  = {

  proc kg_derand(seed : W8.t Array32.t * W8.t Array32.t) :
        pkey * (skey * pkey * W8.t Array32.t * W8.t Array32.t) = {
    kgs <- seed1; z <- seed2;
    (pk,sk) <@ Kyber(HS,XOF,PRF,O).kg_derand(kgs);
    hpk <@ KemH(O).pkH(pk);
    return (pk, (sk,pk,hpk,z));
  }

  proc enc_derand(pk : pkey, prem : W8.t Array32.t) :
      ciphertext * W8.t Array32.t = {
    m <@ KemG(KemHS).sample(prem);
    hpk <@ KemH(O).pkH(pk);
    (_Kt,r) <@ KemH(O).g(m,hpk);
    c <@ Kyber(HS,XOF,PRF,O).enc_derand(pk,m,r);
    hc <@ KemH(O).cH(c);
    _K <@ KemH(O).kdf(_Kt,hc);
    return (c,_K);
  }

  proc dec(cph : ciphertext, sk : skey * pkey * W8.t Array32.t * W8.t Array32.t) :
      W8.t Array32.t = {
    (skp,pk,hpk,z) <- sk;
    m <@ Kyber(HS,XOF,PRF,O).dec(skp,cph);
    (_Kt,r) <@ KemH(O).g(oget m,hpk);
    c <@ Kyber(HS,XOF,PRF,O).enc_derand(pk,oget m,r);
    hc <@ KemH(O).cH(cph);
    if (c = cph) { _K <@ KemH(O).kdf(_Kt,hc); }
    else { _K <@ KemH(O).kdf(z,hc); }
    return _K;
  }
}.
```

Figure 3.2: KYBER EASYCRYPT Specification: IND-CCA KEM

some features that are not relevant for KYBER768, namely noise sampling and compression algorithms that are only used for other KYBER parameter sets[8]. We will gradually introduce the various definitions and intermediate results leading to the full code in Figures 3.2 to 3.5. This section was written so that it is easy to match our code to the pseudocode specification of KYBER in [27, Sec. 1].

3.2.1 Algebraic Structure

Operations over values in field $\mathbb{F}_q$. We begin by fixing the prime $q = 3329$ and instantiating the EASYCRYPT theory `ZModField` with this parameter to obtain type `Fq` that represents elements in the finite field $\mathbb{F}_q$. We extend this theory with an operator `as_sint` that maps elements in `Fq` to integers in the range $[-(q-1)/2, (q-1)/2]$ and define compression and decompression operators as follows:

[8] Extending the specification for other variants is straightforward.

```
op as_sint(x : Fq) = if (q−1) / 2 < asint x then asint x − q else asint x.
op compress(d : int, x : Fq) : int = round (asint x ∗ 2^d /ℝ q) % 2^d.
op decompress(d : int, x : int) : Fq = inFq (round (x ∗ q /ℝ 2^d)).
```

Here, asint is the operator that takes a field element and returns an integer in the range $[0, q)$. We use $/_{\mathbb{R}}$ to denote real division, use / for integer division and % for the mod operation. round denotes rounding to the closest integer with ties being rounded up, as indicated in the KYBER specification. Compression and decompression map field elements into integers in the range $[0, 2^{\mathsf{d}})$ and back, with a loss that is captured by the following lemma stated and proved in EASYCRYPT.

```
op compress_err(d : int, c: Fq) : Fq = decompress d (compress d c) − c.
lemma compress_err_bound (c:Fq) d :
        0 < d =>2^d < q => | as_sint (compress_err d c) | ≤ round (q /ℝ (2^(d+1))).
```

We pave the way to connecting the KYBER specification to Jasmin code by giving alternative definitions of compress and decompress—these match what is done in code using machine instructions—and prove that these are equivalent to the specifications above. The code computes compress using compress_alt for $d = 1$ and $d = 4$, and it uses compress_alt_large for $d = 10$.

```
op compress_alt(d : int, c : Fq) : int = (asint c ∗ 2^d + (q+1)/2) ∗ (2^28/q) / 2^28 % 2^d.
op compress_alt_large(c : Fq) : int = (asint c ∗ 2^10 + (q+1)/2) ∗ (2^32/q) / 2^32 % 2^10.
op decompress_alt(d : int, c : int) : Fq = inFq ((c∗q + 2^(d−1)) / 2^d).
```

Using these operators we can also capture the encoding/decoding of bits as finite-field elements and relate the semantics of these operations to that of the compression and decompression operators, as is done in the KYBER spec.

```
op b_encode(b: bool): Fq = if b then inFq ((q+1)/2) else inFq 0.
op b_decode(c: Fq) : bool = ! |as_sint c| < q / 4 + 1.
lemma b_encode_sem c : b_encode c = decompress 1 (if c then 1 else 0).
lemma b_decode_sem c : compress 1 c = if b_decode c then 1 else 0.
```

The next step is to specify the distributions over field elements that are used by KYBER. We define the Binomial distribution and the uniform distribution over field elements by declaring them as follows, and then deriving auxiliary lemmas that describe the mass function of each of them in an explicit way (this is important for proving correctness of the sampling procedures used by the implementation). We omit the details for brevity.

```
op dshort_elem : Fq distr = dmap (dcbd 2) inFq. (* binomial distribution *)
op duni_elem : Fq distr = DZmodP.dunifin. (* uniform distribution *)
```

The ring $\mathcal{R}_q$, matrices and vectors. At this point we can formalize the ring $\mathcal{R}_q$ of polynomials over which KYBER operates. We start by defining a concrete representation of these polynomials using arrays of size 256 of field elements. As in the KYBER specification, we will reuse this representation for ring elements in the NTT domain.

```
proc kg_derand(seed: W8.t Array32.t)
                              : pkey * skey = {
  (rho,sig) <@ G(HS).sample(seed);
  _N <- 0;
  i <- 0;
  while (i < kvec) {
    j <- 0;
    while (j < kvec) {
      XOF(O).init(rho,W8.of_int j,W8.of_int i);
      c <@ Parse(XOF,O).sample();
      a[(i,j)] <- c;
      j <- j + 1;
    }
    i <- i + 1;
  }
  i <- 0;
  while (i < kvec) {
    c <@ CBD2(PRF).sample(sig,_N);
    s <- set s i c;
    _N <- _N + 1;
    i <- i + 1;
  }
  i <- 0;
  while (i < kvec) {
    c <@ CBD2(PRF).sample(sig,_N);
    e <- set e i c;
    _N <- _N + 1;
    i <- i + 1;
  }
  s <- nttv s; e <- nttv e;
  t <- ntt_mmul a s + e;
  tv <@ EncDec.encode12_vec(toipolyvec t);
  sv <@ EncDec.encode12_vec(toipolyvec s);
  return ((tv,rho),sv);
}

proc dec(sk : skey, cph : ciphertext)
                        : plaintext option = {
  (c1,c2) <- cph;
  ui <@ EncDec.decode10_vec(c1);
  u <- decompress_polyvec 10 ui;
  vi <@ EncDec.decode4(c2);
  v <- decompress_poly 4 vi;
  si <@ EncDec.decode12_vec(sk);
  s <- ofipolyvec si;
  mp <- v - invntt (ntt_dotp s (nttv u));
  m <@ EncDec.encode1(compress_poly 1 mp);
  return Some m;
}
```

```
proc enc_derand(pk : pkey, m : plaintext, r : W8.t Array32.t)
                                          : ciphertext = {
  (tv,rho) <- pk;
  _N <- 0;

  thati <@ EncDec.decode12_vec(tv);
  that <- ofipolyvec thati;
  i <- 0;
  while (i < kvec) {
    j <- 0;
    while (j < kvec) {

      XOF(O).init(rho,W8.of_int i, W8.of_int j);

      c <@ Parse(XOF,O).sample();
      aT[(i,j)] <- c;
      j <- j + 1;
    }
    i <- i + 1;
  }
  i <- 0;
  while (i < kvec) {

    c <@ CBD2(PRF).sample(r,_N);
    rv <- set rv i c;
    _N <- _N + 1;
    i <- i + 1;
  }
  i <- 0;
  while (i < kvec) {

    c <@ CBD2(PRF).sample(r,_N);
    e1 <- set e1 i c;
    _N <- _N + 1;
    i <- i + 1;
  }

  e2 <@ CBD2(PRF).sample(r,_N);
  rhat <- nttv rv;

  u <- invnttv (ntt_mmul aT rhat) + e1;
  mp <@ EncDec.decode1(m);

  v <- invntt (ntt_dotp that rhat) + e2
                                + decompress_poly 1 mp;

  c1 <@ EncDec.encode10_vec(compress_polyvec 10 u);

  c2 <@ EncDec.encode4(compress_poly 4 v);
  return (c1,c2);
}
```

Figure 3.3: KYBER EASYCRYPT Specification: IND-CPA PKE

We write explicit formulae for the basic ring operations over this array representation (shown below) and extend the definitions of compression/decompression and binomial/uniform distributions to ring elements by applying them pointwise to each coefficient.

```
module type XOF_t(O : RO.POracle) = {
  proc init(rho : W8.t Array32.t, i j : W8.t) : unit
  proc next_bytes() : W8.t Array168.t
}.

module Parse(XOF : XOF_t, O : RO.POracle) = {
  proc sample() : poly = {
    j <- 0;
    while (j < 256) {
      b168 <@ XOF(O).next_bytes();
      k <- 0;
      while ((j < 256) && (k < 168)) {
        bi  <- b168[k]; bi1 <- b168[k+1]; bi2 <- b168[k+2];
        k <- k + 3;
        d1 <- to_uint bi        + 256 * (to_uint bi1 % 16);
        d2 <- to_uint bi1  / 16 + 16  * to_uint bi2;
        if (d1 < q)                   { aa[j] <- inFq d1; j <- j + 1; }
        if ((d2 < q) && (j < 256)) { aa[j] <- inFq d2; j <- j + 1; }
      }
    }
    return aa;
  }
}.
```

Figure 3.4: KYBER EASYCRYPT Specification: Rejection sampling a uniform element in $\mathcal{R}_q$.

```
type poly = Fq Array256.t.
op zero : poly = Array256.create Zq.zero.
op one : poly = zero[0<-Zq.one].
op (+) (pa pb : poly) : poly = map2 (fun a b : Fq => Zq.(+) a b) pa pb.
op (-) (p : poly) : poly = map Zq.([-]) p.
op ( * ) (pa pb : poly) : poly =
  Array256.init (fun (i : int) => foldr (fun (k : int) (ci : Fq) =>
    if (0 ≤ i - k) then ci + pa[k] * pb[i - k]
    else ci - pa[k] * pb[256 + (i - k)]) Zq.zero (iota_ 0 256)).
```

We prove that these definitions implement the correct operations over $\mathbb{Z}_q[X]/(X^{256}+1)$ as follows. We first instantiate the generic polynomial theory PolyReduce recently added to EASYCRYPT with $n = 256$ and $q = 3329$, to obtain a description of the corresponding mathematical ring and its operations. We then define a bijection between our poly type above and the type of ring elements in the resulting theory, which allows us to easily prove that our ring operations above are correct with respect to the explicit formulae defined by PolyReduce.

We can now define matrices and vectors over ring elements in our representation by cloning the Matrix theory in the EasyCrypt library with the correct dimension kvec=3 for KYBER768, and instantiating the ring operations with those we described above. Note that, when instantiating the Matrix theory, we must prove that the provided operators indeed constitute a ring, which we can do by leveraging results in the PolyReduce theory.

Number Theoretic Transform. Finally, we complete the algebraic definitions by specifying the Number Theoretic Transform (NTT) in EASYCRYPT and

```
op SHA3_PRF : W8.t Array32.t -> W8.t -> W8.t Array128.t.

clone import PRF_DEFS.PseudoRF as PRF_ with
  type K <- W8.t Array32.t,
  op dK <- srand,
  op F <- SHA3_PRF.

module CBD2(PRF : PseudoRF) = {
  proc sample(sig : W8.t Array32.t, _N : int) : poly = {
    bytes <@ PRF.f(sig, W8.of_int _N);
    bits <- BytesToBits (to_list bytes);
    i <- 0;
    while (i < 256) {
      a <- b2i (nth false bits (4*i)) + b2i (nth false bits (4*i+1));
      b <- b2i (nth false bits (4*i+2)) + b2i (nth false bits (4*i+3));
      rr[i] <- inFq  (a - b);
      i <- i + 1;
    }
    return rr;
  }
}.
```

Figure 3.5: KYBER EASYCRYPT Specification: Sampling noise from the Binomial distribution.

proving some fundamental results about this transform that are essential to the security and correctness of KYBER. We define the NTT forward and inverse operations as literal translations of what is stated in the KYBER specifications (including the use of the same representation for inputs, and outputs, and the operation that commutes with the NTT transform for fast multiplication in $\mathcal{R}_q$). We omit here the definition of NTT for vectors and matrices of ring elements, as this is simply the pointwise application of NTT to each component. We use notation $(\cdot)_1$ and $(\cdot)_2$ for the first and second element in a pair.

```
op zroot = inFq 17.
op br = BitEncoding.BitReverse.bsrev 7.

op ntt(p : poly) = Array256.init (fun i => let ii = i / 2 in
  if i % 2 = 0 then ∑_{j=0}^{127} p[2*j] * zroot^{(2*br ii+1)*j}
               else ∑_{j=0}^{127} p[2*j+1] * zroot^{(2*br ii+1)*j}).

op invntt(p : poly) = Array256.init (fun i => let ii = i / 2 in
  if i % 2 = 0 then ∑_{j=0}^{127} inv (inFq 128) * p[2*j] * zroot^{-(2*br j+1)*ii}
               else ∑_{j=0}^{127} inv (inFq 128) * p[2*j+1] * zroot^{-(2*br j+1)*ii}).

op cmplx_mul (a :Fq * Fq, b : Fq * Fq, zzeta : Fq) : Fq * Fq =
  (a2 * b2 * zzeta + a1*b1, a1 * b2 + a2 * b1).

op basemul(a b : poly) : poly = Array256.init (fun i => let ii = i / 2 in
  if i % 2 = 0 then (cmplx_mul (a[2*ii],a[2*ii+1]) (b[2*ii],b[2*ii+1]) (zroot^{(2*br ii+1)}))_1
               else (cmplx_mul (a[2*ii],a[2*ii+1]) (b[2*ii],b[2*ii+1]) (zroot^{(2*br ii+1)}))_2).
```

In Section 3.5.3 we describe the EasyCrypt library that was developed with general results about the NTT transform, which we used to establish the following properties, as well as natural extensions for vectors and matrices of ring elements.

```
op scale(p : poly, c : Fq) : poly = Array256.map (fun x => x * c) p.
lemma ntt_scale p c : ntt (scale p c) = scale (ntt p) c.
lemma invntt_scale p c : invntt (scale p c) = scale (invntt p) c.
lemma nttK : cancel invntt ntt.
lemma mul_comm_ntt (pa pb : poly): ntt (pa * pb) = basemul (ntt pa) (ntt pb).
lemma add_comm_ntt (pa pb : poly): ntt (pa + pb) = (ntt pa) + (ntt pb).
lemma nttZero : ntt KPoly.zero = KPoly.zero.
```

These properties play an important role in the proof of correctness and in the ongoing proof of security, and they include the expected cancellation (invertibility) properties, commuting with ring addition and multiplication. Additionally, the preservation of scaling factors is used to deal with multiplicative terms used in Montgomery representation (see Section 3.4). We note that these are well known results that are taken as a given in the Kyber specification and, to the best of our knowledge, have not been previously formally verified in the context of cryptographic proofs.[9] As stated in the introduction, we believe that these results are of independent interest, since the NTT is pervasively used in post-quantum cryptography and will surely be used in future EasyCrypt developments.

3.2.2 Algorithmic Sampling, Encoding and Decoding

The Kyber specification relies on a number of auxiliary algorithms. We follow the same structure and define them as independent EasyCrypt modules used by the main Kyber specification in EasyCrypt. In this section we mention SHA3 algorithms several times, when the specification refers to it. By this we mean algorithms in the SHA-3 standard, including the SHA3 hash functions and the SHAKE extendable-output functions. In Section 3.7 we further discuss how our formalization and an ongoing machine-checked security proof of the specification clarify the assumptions that Kyber makes about SHA3 algorithms.

Rejection sampling a uniform ring element. The first auxiliary algorithm used by Kyber is a rejection sampling procedure called Parse, which we give in

[9] In independent works [137, 125] have also looked at the NTT transformation. In [137] the specification of the transform is similar to the one we use; the authors use this to show that the Kyber specification is *cryptographically* a correct KEM and, for this, the authors need to prove a set of properties about the NTT that is close to what we also prove. The specification used in [125] is the mathematical view based on the Chinese Remainder Theorem, rather than the convolution given in the Kyber specification. This specification is used to show correctness of the NTT AVX2 computations, something that we also do as an intermediate result. Our results could be formally related to those in [137, 125] by showing that the NTT specifications are the same.

Figure 3.4. The algorithm is parametric on an Extendable-Output Function (XOF), which must be initialized prior to the execution of Parse, and is then used by this algorithm to obtain as many random bytes as needed to complete the 256 coefficients that make up an element in $\mathcal{R}_q$. We note that in Kyber the output of this procedure is interpreted as being in the NTT domain, which has implications in various parts of the correctness and security proofs. We model the XOF as producing a sequence of byte blocks of length 168 (each call to next_bytes is the next batch of XOF output bytes), since this matches the way Kyber uses SHAKE-128 for this purpose.

Using SHA3 to hash, smooth randomness, and sample noise. Kyber uses SHA3 algorithms over strings of different input lengths to implement the various hash functions required for the KEM (Fujisaki-Okamoto-like) construction. Our specification clarifies exactly how the SHA3 algorithms (a combination of SHA3-256 and SHAKE-256) are being used to "instantiate" these hash functions, as discussed in Section 3.7.

Kyber also uses SHA3 algorithms to expand the raw randomness fed to key generation into two 32-byte seeds, one for the matrix rejection sampling, and another to be used as PRF key in the noise sampling procedure above. Here SHA3-256 and SHA3-512 are used as pseudorandom generators (PRG), referred in the spec as G. Since this PRG maps fixed-length input strings to fixed-length output strings, we model it as a PRF that can be evaluated on a single point.

Finally, Kyber relies on SHA3 algorithms to sample pseudorandom ring elements of small norm (and vectors thereof). The procedure used to sample ring elements is specified in Figure 3.5, which exactly matches the Kyber specification for the concrete parameter of $\eta = 2$, which is used by the implementation of Kyber768. Here we model SHAKE-256 as a Pseudo Random Function (PRF), using the corresponding EasyCrypt library and an operator SHA3_PRF.

In the proofs we describe in this paper we do not assume any security semantics to these uses of SHA3; this will be subject of a follow-up ongoing work to prove the security of the Kyber specification and, by transitivity, security of the implementations. Instead, we simply assume that the Kyber specification is using SHA3 algorithms in exactly the same way that the implementation is using SHA3 algorithms and, prove, under this assumption, that the implementation is functionally correct with respect to the specification. We note that some of the implementations of SHA3 algorithms that we use in the development have been proved correct in [17], but we do not establish a formal connection between the two verification results. This means that, for the time being, correctness of the SHA3 library is an assumption in our proof.

Encoding and Decoding. The last auxiliary module is called EncDec and it specifies various forms of serializing and de-serializing keys, messages and ciphertexts to/from byte arrays. We specify these operations by defining the canonical bijection toipoly/ofipoly between (possibly compressed) ring elements

```
type ipoly = int Array256.t.
op toipoly(p : poly) : ipoly = map asint p.
op ofipoly(p : ipoly)  : poly = map inFq p.

module EncDec = {

  proc decode4(a : W8.t Array128.t) : ipoly = {
    i <- 0;
    while (i < 128) {
      r[i*2+0]  <- to_uint a[i] % 16;
      r[i*2+1]  <- to_uint a[i] / 16;
      i <- i + 1;
    }
    return r;
  }

  proc encode4(p : ipoly) : W8.t Array128.t = {
    j <- 0; i <- 0;
    while (i < 128) {
      fi <- p[j]; j <- j + 1;
      fi1 <- p[j]; j <- j + 1;
      r[i] <- W8.of_int (fi + fi1 * 2^4); i <- i + 1;
    }
    return r;
  } (···) }.
```

Figure 3.6: Imperative specification of encoding/decoding.

and arrays of integers representing the polynomial coefficients, and a number of procedures that encode/decode polynomials that have been previously compressed to varying lengths. We show in Figure 3.6 the case of encoding a polynomial that has been compressed to 4-bits per coefficient into 128 bytes.

These encoding/decoding procedures are specified in imperative form, with a loop structure and output/input byte array types that simplify the connection to the implementation and are also more readable. Our choice here is motivated by the fact that we use fixed-sized arrays for the encoded bytes, which means that we must independently specify an explicit encoding/decoding procedure for each compression size.

In the Kyber specification [27], decoding is defined by first mapping the compressed polynomial to a bit string (BytesToBits), which is subsequently partitioned into the various coefficients. We also formalize this version of the operation for completeness.

```
op BytesToBits(bytes : W8.t list) : bool list = flatten (map W8.w2bits bytes).
op decode(l i : int, bits : bool list) =∑_{j=0}^{l} b2i (nth false bits (i*l+j)) *2^j.
op sem_decode4(a : W8.t Array128.t) : ipoly =
     Array256.init (fun i => decode 4 i (BytesToBits (to_list a))).
```

The Kyber specification [27] leaves encoding implicitly defined as the inverse of the decoding operation. We rely on a new EasyCrypt feature to lift our imperative versions of the encoding and decoding operations to functional operators, prove that decoding matches the Kyber specification, and finally prove that encoding inverts decoding. We discuss this in more detail in Section 3.3.2.

The full specification. The code given in Figures 3.2 to 3.5 and the code snippets we provide throughout this section closely match the pseudo-code and definitions in the Kyber specification [27]. As discussed in the introduction, this is relevant as this specification is the centerpiece of our effort that all formal-verification results refer to. Hence, their applicability to Kyber itself depends

on how faithfully it matches what is written in the handwritten proposal. For this reason, our goal was to simplify the task of a human checking by inspection that the EC specification we described in this section matches the hand-written description[10].

3.3 Our toolchain: Jasmin and EasyCrypt

3.3.1 Jasmin before and after this paper

Jasmin is a programming language and a set of accompanying tools designed to allow the efficient implementation and the formal verification of cryptographic primitives. It empowers programmers with precise control over low-level features of the assembly code produced by the Jasmin compiler, such as instruction selection and register spilling. Yet, the language provides high-level abstractions such as functions and arrays that help structuring programs and proofs. For instance, arrays have a functional semantics and functions are call-by-value, vastly simplifying modular reasoning. In accordance with the high-efficiency objective, most abstractions are eliminated at compile-time; the compiler checks that arrays are used linearly, which in turn allows to modify them in-place and ensure that there are no run-time copies.

Before it was improved with the extensions described in this work, the Jasmin compiler had a few limitations. First of all, running a computation on a slice of an array (i.e., a contiguous region within an array as opposed to the full array) was very tedious and pollutes the proofs. Second, the compiler inlined all functions at every call-site, which dramatically increased the code size. For instance, the Jasmin implementations of the SHA3 algorithms from previous work [17] are already quite large (hundreds or thousands of assembly instructions, depending on the parameters). The Kyber implementations rely on this primitive for various purposes, such that inlining it at every call-site would inevitably result in an unaffordable code blow-up.

To overcome these limitations and allow the efficient implementation of large routines such as Kyber, the Jasmin infrastructure has been extended with three new features described below: local functions, pointers and sub-arrays, and a randombytes system call.

Local functions. Jasmin programs now are made of functions: at the source level, they now come in three flavors: export functions, inline functions, and local functions. Export functions correspond to entry-points, meant to be called from other programs. Therefore they must comply, after compilation, with the targeted ABI. In particular, the number of arguments and returned values is limited and callee-saved registers must be preserved. Moreover, their argument and return types are limited to machine words. At the other end of

[10]The specification of Kyber described in Section 3.2 can be found in our artifact in the file `proof/spec/Kyber.ec` starting from line 1193.

the spectrum, inline functions disappear early in the compilation process. They thus enjoy a lot of flexibility without incurring any run-time cost: they can take any number of arguments of any types, including arrays and return as many as necessary.

Finally, local functions are kept as separate entities after compilation. Yet, they are not exposed to external users and therefore need not comply with any convention. For instance, they can return many results and the programmer chooses whether the return address is passed through the stack memory or through a register. Interestingly, there is no fixed list of callee-saved registers, i.e., registers whose contents must be preserved by function calls. This ensures that there will be no hidden cost to using a function. This is made possible by a rather unique method to register allocation: all functions are compiled together and all their reg local variables are allocated simultaneously.

Arrays, sub-arrays and implicit pointers to stack. In Jasmin, each variable is labeled either reg or stack, specifying that the compiler must allocate it in a register or on the stack respectively. This is the case also for arrays. reg arrays are a convenient way to manipulate several reg variables together. They are compiled to individual registers. stack arrays are allocated as contiguous memory zones in the stack. The compiler, at the stack allocation pass, decides on a shape for the stack, and replaces accesses to arrays with memory accesses at positions relative to SP, the stack pointer. For instance, if array b is allocated on the stack at address SP + 5, an access b[r+3], where r is a register, is compiled to a memory access at address SP + r + 8.

A new feature that represented a major change to the compiler was allowing local functions to accept stack arrays as arguments. This implied a redesign of the stack allocation pass: there are now arrays whose address is not known at compile time. To achieve that, we impose that arrays in the stack are taken by local functions using a new type called reg ptr. At the source level, such an array behaves as the other arrays. Jasmin semantics makes no distinction between arrays of different kinds, thus reasoning is not more complex than before. However reg ptr is compiled as a pointer, to a register holding its address. The stack allocation pass checks that this transformation is valid, meaning that all modifications to the array are done in place. If an array is changed in the body of a local function, it has to be returned to abide by the high-level semantics of Jasmin. This spurious copy will be eliminated during stack allocation. Since reg ptr arrays are actually registers, we need a way to let the user spill them if needed. For that matter, we introduced stack ptr array.

In the version of Jasmin prior to this work, arrays could be passed as arguments to local functions, but there was no way to call a function with only part of the array without making an explicit copy. To make the best use of local functions, we allow the user to take slices of arrays called sub-arrays. This allows to call a local function on a small part of a large array. The starting position and length must for now be known at compile time. If the array is a reg ptr, this is compiled as pointer arithmetic.

Note that the features described in this section are the only facility given to the user to manipulate (indirectly) pointers. The fact that the user never gets direct access to pointers ensures that formally reasoning on the source remains tractable.

Randombytes. Many cryptographic routines process random data. A common practice is to receive randomness as input. However, Kyber (as well as other post-quantum schemes) uses a rejection sampling procedure which may consume an unbounded stream of random data. To overcome this limitation it proves convenient to sample fresh randomness on demand. To this end there is now a randombytes interface in Jasmin. The usual method of taking random data as input remains of course available. Moreover, the user of the Jasmin program is responsible for implementing this interface: no actual implementation is needed before link time. Possible implementation methods include asking the OS through a system call, or calling some dedicated library, e.g., to access a hardware-security module.

Semantically, this is modeled as an unspecified deterministic function in a state monad. This ensures that the compiler correctness theorem applies seamlessly. The EasyCrypt formalization uses a slightly different view: the EasyCrypt code is parameterized by a module providing the implementation of this system call. Said implementation can indeed be stateful. The functional correctness of the Kyber implementations can be established for any instantiation of the parameter, since they will match the semantics of the specification for any input random bytes. Security analysis (ongoing work) however will need to make assumptions about the distribution.

More generally, there is a general facility to call external functions from Jasmin programs. Currently only randombytes is available through this mechanism but, should the need arise, other operations can be seamlessly added.

3.3.2 EasyCrypt and extensions added in this work

EasyCrypt is a proof assistant geared towards proving the correctness and security of implementations of cryptographic primitives. The Jasmin compiler can output either assembly or EasyCrypt code with semantics equivalent to its input. Properties about this EasyCrypt code can then be proven in EasyCrypt using one of the variants of Hoare logic it features:

- HL is a deterministic Hoare logic that allows to prove result on deterministic programs. Statements in this logic are of the form **hoare** [M.foo : P ==> Q], where M.foo is a procedure in module M, P is a precondition and Q is a post condition.
- pHL is a probabilistic Hoare logic, that allows to reason about the probability of occurrence of events in potentially probabilistic programs. Statements in this logic are of the form **phoare** [M.foo : P ==> Q] = p, where

p is now the probability that Q holds in the post-state. In the current paper we typically have p=1 and the pHL statement can be proved by first establishing that the program always terminates, and then proving the statement using HL.

- pRHL is a probabilistic relational Hoare logic, that allows to prove relational results about the lock-step execution of a pair of programs. Statements in this logic are of the form **equiv** [M.foo ~ N.foo : rP ==> rQ], where rP and rQ are now relational pre- and post- conditions that can refer to the initial and final states of both procedures.

In particular, pRHL can be used in conjunction with pHL to prove results by *game-hopping*: using a correctness result about a procedure P_1 in pHL and an equivalence result between procedures P_1 and P_2 in pRHL, one can prove a correctness result about P_2. Also, in a deterministic setting, results in HL can be proven using results in pHL, as well as the other way around. The proof of correctness for the Kyber Jasmin code relies on these two mechanisms. In particular, the game-hopping strategy allows us to build intermediate programs between the EasyCrypt code produced by Jasmin and our desired specification. We use it to remove optimizations and to abstract operations on algebraic structures.

EasyCrypt also features a standard library formalizing some basic algebraic structures, which we use to capture the algebraic structure underlying Kyber. This library is based on the EasyCrypt cloning mechanism, originally introduced to manage abstract types in cryptographic primitives. One thing that became clear during this project is that cloning has proven to be a source of complexity and limitations as more algebraic structures were added to the hierarchy, namely polynomial rings, vectors and matrices thereof needed for Kyber. We have communicated this to the EasyCrypt developers and learned that future versions of EasyCrypt will use type classes for this purpose.

The most innovative use of EasyCrypt in this development was, perhaps, the functional correctness of the Number Theoretic Transform (NTT) implementations, which we will describe in detail later in the paper. However, a number of enhancement to EasyCrypt were created to support this development, which we summarize next.

Additions to the standard library. Kyber is the first cryptographic primitive studied using EasyCrypt to involve algebraic structures such as polynomials arithmetic, modular rings and ring quotients. As a result, many parts of the standard library had to be created or improved to lay foundations underlying the Kyber specification. These included results about lists, bit-vector manipulations, ring quotients, logarithm and p-adic valuations.

Automatic inference of functional specifications. One of the tedious aspects of some EasyCrypt proofs arises from the fact that some parts of the proofs are easier to do with functional specifications, whereas other parts are

easier to do with imperative specifications. The sweet spot usually resides in using both approaches at the same time, which means writing both specifications by hand and proving that they are equivalent. In Kyber, a number of simple procedures are used to encode and decode data into byte arrays, so applying this approach would be quite time consuming. To avoid this, a new mechanism was developed that can automatically infer a functional specification from an imperative program, provided that the latter has as reasonably simple control flow.

Dealing with nested for loops in EasyCrypt. The reference implementation of the NTT in Kyber involves multiple nested for loops whose formal study is tedious in EasyCrypt. Indeed, for dealing with loops, EasyCrypt only comes with a restricted set of general purposes proof rules that trade expressiveness for ease of use. In consequence, we designed a new library that recasts the aforementioned general purpose rules as simplest ones but that only apply to more restricted forms of while loops, as the ones that can be found in Kyber. This library targets the different flavors of EasyCrypt's Hoare logic and has proven particularly helpful in *game-hopping* based proofs on programs that contain multiple nested for loops.

3.4 Implementation

This paper presents two different implementations of Kyber768, a reference implementation and an optimized implementation using AVX2 vector instructions. Those implementations follow largely the structure and implementation strategies of the `ref` and `avx2` implementations by the Kyber team[11], which are also included in the submission package to NIST and integrated into PQClean[12].

Reference implementation. The structure of the reference implementation largely follows the specification. On the lowest level they implement arithmetic in $\mathbb{F}_q$, i.e. on integers modulo the prime q; these are represented as signed 16-bit integers. The most interesting routines on this level are Montgomery and Barrett reductions. The Montgomery reduction takes as input a signed 32-bit integer a and produces a signed 16-bit output r with $r \equiv a \cdot 2^{16} \pmod{q}$. This function does not actually work for arbitrary inputs and is also not producing fully reduced results: more specifically, it requires the input to be in $\{-q \cdot 2^{15}, \ldots, q \cdot 2^{15} - 1\}$ and guarantees that the output is in $\{-q+1, \ldots, q-1\}$. Barrett reduction takes an arbitrary signed 16-bit integer a as input and produces $r \in \{-(q-1)/2, \ldots, (q-1)/2\}$, such that $r \equiv a \pmod{q}$.

On top of this is arithmetic in $\mathcal{R}_q$, i.e., on univariate polynomials over $\mathbb{F}_q$ modulo $(X^{256}+1)$. Addition and subtraction in $\mathcal{R}_q$ are simply coefficient-wise

[11] See `https://github.com/pq-crystals/kyber`.
[12] See `https://github.com/PQClean/PQClean`

addition and subtraction of arrays of 16-bit integers. The implementation follows a lazy-reduction approach here and does not apply modular reduction after addition and subtraction. The more interesting routines on polynomials are forward and inverse NTT, and "pointwise" multiplication of polynomials in NTT domain. The forward NTT takes an array of 256 coefficients in normal order and produces the output inplace in bit-reversed order. It uses 7 layers of Cooley-Tukey butterfly operations. Twiddle factors are precomputed in Montgomery and multiplication by those twiddle factors is followed by Montgomery reduction, bringing the result back to normal domain. Addition and subtraction inside the butterflies is not followed by modular reduction, i.e., also here the implementation makes use of the lazy-reduction optimization. The inverse NTT takes input in bit-reversed order and uses 7 layers of Gentleman-Sande butterflies to produce output inplace in normal order. Multiplication by twiddle factors is again using Montgomery arithmetic. The result of addition inside the butterflies is reduced using the Barrett reduction routine. The result of subtraction is again left unreduced. As the NTT in Kyber is incomplete (see [27, Sec. 1.1]), the pointwise multiplication of polynomials in NTT domain corresponds to 128 multiplications of 2-coefficient polynomials modulo $X^2 - \zeta^i$, where ζ^i are precomputed powers of $\zeta = 17$, the smallest 256-th root of unity modulo q. This operation is computed through Montgomery multiplications and additions that again omit modular reduction. Note that Montgomery reduction moves resulting coefficients to "inverse Montgomery domain", i.e., they get multiplied by a factor of 2^{-16}. This factor is removed again at the very end of the inverse NTT. For the public key, which is transmitted in NTT domain, the factor is removed in a dedicated function that multiplies each coefficient by 2^{32} mod q and then applies Montgomery reduction.

Also on this layer are functions to sample polynomials with coefficients that are uniformly distributed modulo q and to sample noise polynomials with coefficients that follow a centered binomial distribution with parameter $\eta = 2$. The former is using simple rejection sampling on output of SHAKE-128. The latter is a bit more involved and treats 32-bit words (in the C reference implementation) or bytes (in the Jasmin reference implementation) as vectors of 2-bit integers to compute Hamming weight and subsequent subtraction. Finally, the implementation of $\mathcal{R}_q$ includes functions for serialization and deserialization and for "LWR-style" rounding to a smaller, power-of-two modulus to compress ciphertexts. For the reference implementations, these functions are rather straightforward translations of the specification into C respectively Jasmin.

The highest level of arithmetic implements operations on vectors and matrices over $\mathcal{R}_q$. These functions are all rather straightforward loops over calls to functions performing arithmetic in $\mathcal{R}_q$. These functions are first combined to implement the Kyber.CPAPKE passively secure public-key encryption scheme and then, together with multiple functions derived from Keccak (SHA3-256, SHA3-512, SHAKE-128, SHAKE-256) to build the full IND-CCA2 secure KEM Kyber.

AVX2 implementation. As most optimized implementations, the main idea of the AVX2 implementation code is to identify the functions that consume most of the CPU cycles in the reference implementation and, one-by-one, replace those by more optimized versions. One might think that verification "simply" proves equivalence between the reference and the optimized implementations function-by-function and that this verification would mostly mean showing equivalence of longer loops over scalar operations and shorter loops over vector operations. While this is true for some simple functions like polynomial addition and subtraction, the differences are much more substantial for the bulk of the AVX2 implementation. The most interesting differences between the reference and the AVX2 implementation are the following:

- Whenever possible, the AVX2 implementation uses a Keccak implementation that processes 4 independent streams of input and output in parallel. Using this more efficient 4× parallel implementation requires the high-level code to batch calls; the vectorization is thus not a local change encapsulated inside a low-level function anymore.

- The AVX2 implementation of the NTT produces output in a "permuted bit-reversed" order; this order is also used by the pointwise multiplication, so all polynomials in NTT domain use a different ordering of coefficients in the AVX2 implementation compared to the reference implementation. As the public key of Kyber is transmitted in NTT domain, serialization of the public key includes a permutation to move to standard bit-reversed order.

- The AVX2 NTT code is also an illustrative example of where the aggressive optimization strategies, together with the willingness to maximize the impact of the available AVX2 registers and instruction-set, often leads to ingenious but contrived code, intertwining arithmetic and bitwise operations. Correctness verification is challenging, since one needs to keep track of what is happening at each step with significant detail, but the deep understanding of the underlying arguments occasionally suggested tweaks for additional speed-ups that were incorporated into the code.

- The AVX2 modular reductions are subtly different, so they produce output that is equivalent modulo q, but not equal to those produced by the reference implementation. This also means that the whole lazy-reduction strategy of the AVX2 implementation is different, which requires updates to the range analysis to make sure that no overflows occur.

- The compression function for vectors of polynomials uses a different algebraic expression to compute the rounding to the smaller power-of-two modulus. More specifically, while the reference implementation computes rounding from $q = 3329$ to $q' = 1024$ as `((1024*i + (Q-1)/2)/Q) & 1023`, the AVX2 implementation uses `(8*V*i - (i+15) + (1 << 18)) >> 19`,

with the value `V` set to 20159. The equivalence of these two expressions for all relevant inputs is in principle not hard to prove, but the situation is further complicated by the way how results of vectorized multiplications are split into low and high parts of the result. In other words the complication is extracting this algebraic expression from the code in the first place.

- Finally, the rejection sampling to generate the matrix $\mathbf{A}$ is using a rather involved vectorization approach inspired by the techniques introduced for NewHope in [109, Sec. 4.2]. This routine is the only part of the AVX2 implementation that is not yet proven correct; this is why Section 3.6 reports performance results for a fully optimized (with this involved AVX2-based sampling) and a fully proven (with the sampling from the reference implementation) implementation. This faster rejection sampling routine is challenging to verify because it takes a totally different approach to generating the necessary coefficients than the reference implementation: rather than going sequentially through the output bytes produced by the XOF, it generates a large number of candidates and checks their bounds in parallel, to compute what intuitively is a bit map of the good elements; the code then uses this information to *pack* these good elements in a contiguous memory region, and then takes the necessary prefix. Since the used set of SIMD instructions and optimization thechniques have little in common with the rest of the code, verifying this part of the implementation represents significant additional effort.

3.5 Proof of functional correctness

The strategy we adopted for the functional correctness proof divided the effort into a first phase, in which we constructed a proof for the reference implementation, and a second phase where we addressed the AVX2 implementation. The first phase included the development of a number of EasyCrypt libraries to deal with various data structures and operators used by the EasyCrypt code extracted from the Jasmin implementations, namely distributions over arrays, conversions between arrays of field elements and their representations, etc. One important library that may be reused in future developments establishes results for field elements in Montgomery representation, including not only the expected congruence relations, but also the ranges of the representations of these elements when stored as microprocessor words by the implementation. For the latter, a formalization and proof of the Barrett reduction operation correctness. Another one establishes results for the NTT, which we will describe in detail at the end of this section.

3.5.1 Verifying the Reference Implementation

The proof for the reference implementation follows the hierarchical structure of the code described in Section 3.4. We present the results we establish in a bottom-up order.

Field operations. The lowest-level results concern field-level operations for addition, multiplication, subtraction, and Barrett reduction. The following example shows the proved lemma for field multiplication in Montgomery representation.

```
lemma fqmul_corr _a _b : phoare [ M.__fqmul :
  W16.to_sint a = _a  ∧  W16.to_sint b = _b ==> W16.to_sint res = SREDC (_a * _b)] = 1.
```

The lemma states that the Jasmin function `__fqmul`, given two 16-bit words `a` and `b`, produces a result `res` with probability 1, meaning that it always terminates, satisfying the expected relation to the input parameter values: the value stored in the output 16-bit word, when seen as a signed integer, contains the output of a specification operator `SREDC` computed over the inputs. The operator is defined in the `Montgomery` theory as follows:

```
op SREDC (a: int) : int =
  let u = smod (a * qinv * R) (R^2) in
  let t = smod (a - u / R * q) (R^2) in smod (t / R % (R^2)) R.

lemma SREDCp_corr a:
  0 < q < R / 2 =>
  -R / 2 * q ≤ a < R / 2 * q => -q ≤ SREDC a ≤ q  ∧ SREDC a % q = (a * Rinv) % q.
```

Note that the operator semantics is given by the above lemma both in terms of its algebraic meaning and input/output value ranges. (Here `R` is 2^{16}, the range of 16-bit words.) Note also that this lemma imposes a precondition on the ranges of input values, which must be met by caller functions.

At the field level we also state and prove auxiliary lemmas that relate some sequences of processor operations used by higher-level functions to the field element compression operators introduced in Section 3.2. These are then used in the proofs of correctness for ring element and vector operations.

Ring (polynomial) operations. Operations over ring elements come in three flavors: i. those that follow from the pointwise application of some operation to each coefficient; ii. those that involve reading and writing polynomials from memory; and iii. NTT-related operations. The first type of operations includes addition, subtraction, Barrett reduction, etc. The correctness proofs for these Jasmin functions are straightforward loop arguments that rely on the lower level results described above and stated similarly to the `phoare` judgment given as an example for field multiplication.

The operations involving memory access typically perform compression followed by memory write operations, or memory read operations followed by decompression. The values read from/written to memory are decoded/encoded into byte arrays, as described in Section 3.2. This motivates a different form of functional correctness statement, which we illustrate with the Jasmin function that writes a compressed ring element to memory.

```
lemma poly_compress_corr _a _p mem :
  equiv [ M._poly_compress ~ EncDec.encode4 : pos_bound256_cxq a\{1\} 0 256 2  ∧
          lift_array256 a\{1\} = _a  ∧ p\{2\} = compress_poly 4 _a  ∧
          valid_ptr _p 128  ∧ Glob.mem{1} = mem  ∧ to_uint rp\{1\} = _p    ==>
          lift_array256 res\{1\} = _a  ∧ pos_bound256_cxq res\{1\} 0 256 1  ∧
          touches mem Glob.mem\{1\} _p 128  ∧ load_array128 Glob.mem\{1\} _p = res\{2\}].
```

The above lemma establishes the equivalence between the specification of the encoding procedure EncDec.encode4 and the _poly_compress Jasmin function. It states that, when the input parameter a{1} to the Jasmin function contains the representation (the lift_array256 operator) of a ring element _a with coefficients in the $[0..2q)$ range (pos_bound_256_cxq predicate), and if it starts executing in some initial memory mem, then the final state of the memory will satisfy two properties: i. memory is only modified in a specific region identified via the touches predicate; and ii. reading from that region yields the correct encoding result (this is stated by fixing the input p{2} to EncDec.encode4 on the right to be the result of compressing ring element _a. Expressing correctness in this way has two advantages: i. we can match the control flow of the encoding specification to the control flow of the implementation (see Section 3.2) and rely on relational Hoare logic to avoid dealing with closed forms for the encoding mechanisms in this part of the proof and; ii. we can directly plug-in these **equiv** results into the top level equivalence proof for the IND-CPA encryption scheme.

Finally, the most challenging steps in the proof of ring operations concern the three functions used for NTT-based multiplication: NTT, inverse NTT and pointwise multiplication of degree one polynomials. We prove correctness of the NTT and inverse NTT functions using the same sequence of steps. We first show that the NTT function does not introduce computation errors at the representation level; to this end, we define an EasyCrypt procedure that has the same control flow as the Jasmin implementation, but performs all operations in $\mathbb{F}_q$. This is expressed as the following equivalence in EasyCrypt between our idealized NTT implementation NTT.ntt and the actual extracted code _poly_ntt.

```
equiv ntt_correct_aux :
  NTT.ntt ~ M._poly_ntt :
  r{1} = lift_array256 rp{2}  ∧ zetas{1} = zetas  ∧ signed_bound_cxq rp{2} 0 256 2  ==>
    res{1} = lift_array256 res{2}  ∧ pos_bound256_cxq res{2} 0 256 2.
```

This equivalence imposes a contract on the ranges of input/output representations for the implementation, and it shows that our idealized code, when called with the ring-element represented by the input to the Jasmin code,

and a set of global constants (zetas) used by the Jasmin code, computes the same result over the ring.

We then prove that the global constants used by the implementation are correct representations of the necessary roots of unity, and we prove that our idealized algorithm computes the mathematical NTT operator. This part of the proof relies on an NTT library developed for this project, which we describe in more detail in Section 3.5.3. The final statement looks as follows:

```
lemma ntt_correct _r :
  phoare[ M._poly_ntt :
  _r = lift_array256 rp  ∧ signed_bound_cxq rp 0 256 2   ==>
    ntt _r = lift_array256 res  ∧ pos_bound256_cxq res\{2\} 0 256 2] = 1.
```

The correctness proof for the _poly_basemul and NTT inverse operations come with an interesting twist. Rather than computing the pointwise multiplication of the degree one polynomials, the former actually uses the _fqmul field multiplication operation in a way that leaves an extra Montgomery inverse multiplicative term in the result. Conversely, the NTT inverse operation produces a result that carries an uncancelled Montgomery multiplicative term. This is achieved by introducing an additional Montgomery multiplicative term in the precomputed constants, and therefore has no impact on performance. Since the NTT transform preserves multiplicative scalars (a property we have proved), the composition of these two functions actually produces a correct result without any spurious terms. In Section 3.5.3 we give some more detail on the EasyCrypt library we developed to support the NTT proofs.

Vector operations. Most operations over vectors do not bring any additional challenges: some of them work coefficient-wise over all the coefficients of all ring elements in the vector, and therefore almost identical to the corresponding ring operations, or they are point-wise operations over ring elements (e.g., vector addition, NTT, etc.) which we simply handle by plugging in the results described above three times. The remaining and most interesting operation is the inner product between two vectors in the NTT domain, which we prove correct by relying on the correctness of the basemul and poly_add functions, to get the following result. (Here again we note the extra Montgomery inverse multiplicative term (inFq 169) that will be subsequently canceled by the inverse NTT operation.)

```
lemma polyvec_pointwise_acc_corr_alg va vb :
  phoare [ M.__polyvec_pointwise_acc :
    nttv va = lift_vector a   ∧ nttv vb = lift_vector b   ∧
    signed_bound768_cxq a 0 768 2   ∧ signed_bound768_cxq b 0 768 2    ==>
      signed_bound_cxq res 0 256 2   ∧
      lift_array256  res = scale (ntt (dotp va vb)) (inFq 169)  ] = 1.
```

Top-level correctness. At the top level of the correctness proof reside the lemmas that establish an equivalence between the Kyber specifications of the IND-CPA encryption scheme and the KEM construction described in Section 3.2 and the corresponding Jasmin implementations. The proof of the

```
proc __gen_matrix(seed : W8.t Array32.t, trans : bool) : W16.t Array2304.t = {
  i <- 0;
  while (i < kvec) {
    j <- 0;
    while (j < kvec) {
      XOF(H).init(seed, if trans then W8.of_int i else W8.of_int j,
                        if trans then W8.of_int j else W8.of_int i);
      c <@ Parse(XOF, H).sample();
      a <- (a[i, j <- c])%Kyber;
      j <- j + 1;
    }
    i <- i + 1;
  }
  return unlift_matrix a;
}
proc sample_noise2(noiseseed:W8.t Array32.t) : W16.t Array768.t * W16.t Array768.t * int= {
  _N <- 0;
  i <- 0;
  while (i < kvec) {
    c <@ CBD2(KPRF).sample(noiseseed,_N);
    noise1 <- set noise1 i c;
    _N <- _N + 1; i <- i + 1;
  }
  i <- 0;
  while (i < kvec) {
    c <@ CBD2(KPRF).sample(noiseseed,_N);
    noise2 <- set noise2 i c;
    _N <- _N + 1; i <- i + 1;
  }
  return (unlift_vector noise1,unlift_vector noise2,_N);
}
```

Figure 3.7: Mock matrix and noise generation functions.

KEM implementation correctness follows directly from the fact that the IND-CPA implementation is correct and the various calls to the hash functions in the Fujisaki-Okamoto-like construction are done in the correct sequence. Two additional simple results are needed for comparing two arrays and conditionally copying an array in constant-time.

The proof of the IND-CPA encryption scheme implementation is done in two steps. We first create an EasyCrypt auxiliary implementation of the scheme that is very close to the extracted Jasmin code, bar the fact that we restructure it slightly to better match the structure of the specification and we create additional mock versions of the matrix and noise generation functions, as shown in Figure 3.7. These functions have the same type as the extracted ones, but exactly match the specification modulo the representation of outputs, which here comes as arrays of processor words. As an intermediate step we prove that these functions are equivalent to the corresponding ones in the extracted Jasmin code, assuming that both use SHA3 in the same way: to this end we parameterize the mock implementations in Figure 3.7 with concrete XOF and a PRF modules that call SHA3 code in exactly the same way as the Jasmin implementations do. Looking back at Figure 3.1, this explains why we show the EC specification parameterized by SHAKE-128 and SHAKE-256. In the same figure we also show that we have proved as important side-results that, should

```
lemma kyber_kem_correct_enc mem _ctp _pkp _kp :
  equiv [ Jkem.M(Jkem.Syscall).__crypto_kem_enc_jazz ~
                          KyberKEM(KHS,XOF,KPRF,KHS_KEM,KemH,H).enc_derand:
    valid_ptr _pkp (384*3 + 32) ∧ valid_disj_reg _ctp (3*320+128) _kp (32) ∧
    Glob.mem{1} = mem ∧ to_uint ctp{1} = _ctp ∧
    to_uint pkp{1} = _pkp ∧ to_uint shkp{1} = _kp ∧
    pk{2}_1 = load_array1152 mem _pkp ∧
    pk{2}_2 = load_array32 mem (_pkp + 3*384)
    randomnessp{1} = prem{2} ∧
     ==>
      touches2 Glob.mem{1} mem _ctp (3*320+128) _kp (32) ∧
      let (c,k) = res{2} in
          c_1 = load_array960 Glob.mem{1} _ctp ∧
          c_2 = load_array128 Glob.mem{1} (_ctp + 960) ∧
          k = load_array32 Glob.mem{1} _kp
].
```

Figure 3.8: Example of a top-level correctness result.

the XOF and PRF modules we show above be replaced with purely random coins, then the noise sampling and matrix sampling procedures in the KYBER specification produce the correct output distribution.

The top-level IND-CPA proof then proceeds by connecting our intermediate implementation to the KYBER specification by using the lower level results described above for ring elements and vectors thereof. The proof boils down to guaranteeing that plugging the outputs of one function into the next always satisfies the contracts that constrain the ranges of the representations. When it comes to noise sampling and matrix sampling, we first replace the extracted code with the mock implementation, which allows us to conclude the equivalence proof in a trivial way. An example of a top-level result is given in Figure 3.8 for encapsulation.

In addition to the validity of the input memory regions, the precondition binds these inputs to symbolic constants representing pointers to the input public key provided to the implementations, and specifies that the input public key provided to the specification should match the one stored in the starting memory for the implementation. Furthermore, the random coins provided to both algorithms are also assumed to be the same. The post condition specifies which part of the memory is modified, and states that reading from that memory region correctly recovers the ciphertext and key computed by the specification.

3.5.2 Verifying the AVX2 Implementation

In light of the work described in the previous section for the reference implementation, the process for the AVX2 implementation can also be described as having two parts: 1) showing that AVX2 operations over ring elements and vectors are equivalent to their reference counterparts and 2) proving that this implies that the AVX2 implementations of the IND-CPA PKE and KEM, which call these functions, are equivalent to their reference implementation counterparts. The proof of correctness then follows by transitivity, since we

have already proved that the reference implementation is correct with respect to the KYBER specification. We now describe these steps in more detail.

Top-level Correctness. We prove functional correctness statements for the KEM and IND-CPA encryption scheme constructions that are identical to those given above for the reference implementation. This is natural, since the AVX2 implementation is meant to be a drop-in replacement, whenever the target platform supports the necessary extensions. The proof is by transitivity: we have that the reference implementation is correct with respect to the KYBER specification, so it suffices to prove that the AVX2 implementation is equivalent to the reference implementation. We propagate this transitivity argument using relational Hoare logic down to lower level operations, as far as the structure of the code allows. For example, it is straightforward to prove that the IND-CPA encryption operations are equivalent, assuming that the intermediate level operations over vectors and ring elements on each side are, in turn, equivalent. We end up with a series of proof goals that look like the statement below.

```
equiv reduceequiv :
  Jkem_avx.M.__poly_reduce ~ Jkem.M.__poly_reduce :
    lift_array256 arg{1} = nttunpack (lift_array256 arg{2})  ==>
      lift_array256 res{1} = nttunpack (lift_array256 res{2}) /\
      pos_bound256_cxq res{1} 0 256 2 /\ pos_bound256_cxq res{2} 0 256 2.
```

Here, we have on the left an intermediate-level AVX2 function that performs Barrett reduction of polynomial coefficients and, on the right, the corresponding reference implementation function. Note that we express the input/output relation modulo a permutation nttunpack, since this particular function is executed over ring elements in the NTT domain, which the AVX2 code stores differently (see Section 3.4).

Lower-level equivalences. As one proceeds down the call tree, the code of the AVX implementation becomes significantly different from the reference implementation: there is no-longer a one-to-one correspondence between inner functions and the control-flow is completely different. This is because the optimization based on vector operations relies on a totally different instruction schedule. The most extreme cases are, perhaps, the NTT and inverse NTT operations, where the order in which the computations are performed is indeed totally different: after a first round where the full array is transformed, the algorithm processes the bottom half of the array before moving to the top one.

To prove these lower-level equivalences, we therefore proceed in two steps. We first prove that the AVX2 function satisfies the same contract that we proved for the corresponding reference implementation function. For simple functions, such as addition, subtraction, etc., this task takes similar (small) effort when comparing to the reference implementation counterparts. The only difference lies in the data types and operations used, but the proof steps are essentially the same.

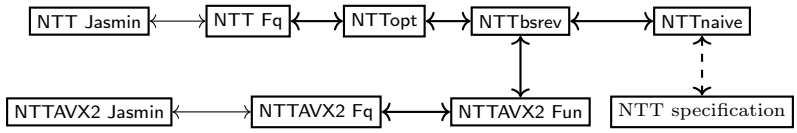


Figure 3.9: Structure of the NTT correctness proof.

For more complex functions the proof effort was sometimes significantly greater, as SIMD operations often require sequences of intricate permutations before a semantically meaningful arithmetic or bitwise operation can be executed. Again, the most challenging cases were the NTT-related operations, where the proof effort was divided into three additional steps: i. defining an intermediate EasyCrypt implementation that performs field operation in the same order as the AVX implementation; ii. proving that this code is equivalent to the AVX implementation by excluding the possibility of representation errors; and iii. proving that the resulting sequence of field operations is indeed computing the NTT as prescribed by the specification. Our NTT library was extended with additional variants of the NTT computation, all proved equivalent to the specification, to facilitate the connection to the AVX implementation, as we describe next.

3.5.3 Verifying the NTT

As described above, to manage complexity, the proof of NTT correctness has two main parts: 1) dealing with the representation of finite field elements and 2) proving that the computations performed over field elements is correct with respect to the specification. Step 1) is carried out as a simple game hop where we replace all the Jasmin register and stack variables that represent group elements, as well as computations over these variables, with computations over $\mathbb{F}_q$. Proving the correctness of this step is done by relying on a series of low level results for computations over $\mathbb{F}_q$.

In step 2), we take care of the most interesting part of the correctness proof: dealing with various optimizations such as the use of an array to store the values of the 256^{th} roots of the unity, the efficient computation of the index used to access this array, and the computation in reverse bit order. The proof is again done by game-hopping and its structure is summed up in Figure 3.9. The thin double arrow represents step 1) discussed above. The thick double arrows correspond to pRHL equivalence proofs. The dashed double arrow corresponds to the final HL correctness proof, where the EasyCrypt program NTTnaive is proven to implement the EasyCrypt functional specification of the NTT, described in Section 3.2. This proof is then lifted to pHL and carried by transitivity to all the other versions of the NTT via the other equivalence proofs. The intermediate NTTs are characterized by the following:

- NTT Fq replaces the calls to auxiliary functions in NTT Jasmin by their mathematical specification
- NTTAVX2 Fq replaces the calls to auxiliary functions in NTTAVX2 Jasmin by their mathematical specification
- NTTopt splits the NTT into several small functions to allow for modular reasoning
- NTTbsrev replaces the incremental computation of the index used to access the array of roots of unity by the computation of it's value from the loop indexes, and the index variable of the innermost loop by it's final value when it's used after the innermost loop
- NTTnaive has it's two innermost loops in reverse bit order when compared to NTTbsrev, and the accesses to the array of roots of the unity are replaced by the computation of the value of the desired root
- NTTAVX2 Fun replaces the two innermost loops by the evaluation of a function, and is designed to be easy to be shown equivalent to both NTTbsrev and NTTAVX2

The most challenging proof was the equivalence between NTTbsrev and NTTnaive, as it is in this transformation that the loops on both sides do not have the same order. As pRHL was made to reason about the lock-step execution of two programs, the invariants used were more complex than in other proofs. The AVX2 NTT implementation, NTTAVX2 Fq, is shown in pRHL to be equivalent to NTTAVX2 Fun, which is itself shown to be equivalent to NTTbsrev. The structure of the correctness proof of the inverse NTT, and the challenges it poses are essentially the same.

3.6 Benchmarks

In this Section we present benchmark results of our Jasmin implementations and compare to the original implementations written in C and assembly. We benchmark on three different generations of Intel CPUs that all implement the AMD64 architecture and support the AVX2 instruction set. Specifically, we benchmark on an Intel Core i7-6500U (Skylake), an Intel Core i7-4770K (Haswell) and on an Intel Core i7-10700K (Comet Lake) CPU. All benchmarks are using only a single core of these CPUs; we followed the standard practice of disabling Hyperthreading and TurboBoost; the C compiler used for all benchmarks was Clang, version 14.[13]

Table 3.1 reports the results of these benchmarks. All cycle counts in this table are the median of 10 000 runs with different inputs. All

[13] clang-14 -march=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE

Table 3.1: Cycle counts for C, C/assembly, and our Jasmin implementations of Kyber768 on different generations on Intel CPUs

Implementation	operation	Skylake	Haswell	Comet Lake
C ref	keygen	200 302	187 172	184 374
	encaps	251 384	242 424	235 714
	decaps	287 724	278 160	272 296
Jasmin ref	keygen	411 676	394 636	384 948
	encaps	488 904	471 680	458 640
	decaps	562 426	534 420	527 266
C/asm AVX2	keygen	49 572	47 280	41 682
	encaps	60 018	62 900	55 956
	decaps	45 854	47 784	43 906
Jasmin AVX2 (fully verified)	keygen	106 578	96 296	93 244
	encaps	119 308	111 536	107 474
	decaps	105 336	98 328	96 564
Jasmin AVX2 (fully optimized)	keygen	50 004	48 800	45 046
	encaps	65 132	63 988	59 496
	decaps	50 340	51 444	48 172

implementations used the same implementation of `randombytes`, which is implemented through the Linux `getrandom` system call. We see that the Jasmin reference implementation is considerably slower than the C reference implementation. This is expected, because the Jasmin compiler, unlike C compilers, does not optimize the code: this is by design the job of the programmer. The goal of this implementation was not to achieve high performance, but rather to facilitate the EasyCrypt proof of correctness.

More relevant is the performance comparison of the optimized AVX2 implementations. We first comment that, for the same C/asm implementation and very similar CPUs, SUPERCOP [46] reports somewhat lower cycle counts. The main reason for this difference is that our benchmarks obtain random bytes through a system call, while SUPERCOP uses a fast userspace randomness generator. We see that the fully optimized Jasmin code is within about 10% of the performance of the optimized C and assembly implementation by the Kyber team. We also see the rather dramatic performance impact of our two implementations of rejection sampling, i.e., the only difference between the fully verified and the fully optimized AVX2 implementations. Clearly this difference motivates further effort on verifying a faster rejection sampling routine, but, as we will discuss in Section 3.7, such a proof might want to wait until EasyCrypt supports a higher degree of proof automation.

3.7 Discussion

Proof effort and the need for automation. The development we present in this paper took almost three years to complete. This was certainly to some extent because of the large number of people involved, their availability to work on the project, and their varying sets of skills and expertise. More importantly though, it was due to the sheer scale of the effort and the fact that different tools needed to be improved in order to tackle the example at hand.

If an equivalent project were to be started now with the same team, e.g., to formally verify the implementation of another lattice-based scheme, the expected development time would be cut down significantly due to the accumulated experience and improved tooling, but it would still require a significant resource investment. An obvious conclusion we draw from this fact is that we need to be able to increase automation in the available tools.

The approach in [125] is promising but, at the moment, there are significant differences in the level of assurance that our work provides in comparison, namely: (1) our coverage of the implementation (excluding SHA3) is complete, whereas in [125] only the NTT transform is covered; and (2) the trusted computing base (TCB) for the tooling in our development is arguably smaller, since EasyCrypt has been around for a long time and is reasonably self-contained, and the Jasmin compiler is verified in Coq. Nevertheless, like in [125], our TCB includes hand-written semantics for assembly operations and the part of the Jasmin compiler that extracts Jasmin code to EasyCrypt. In any case, future research will certainly include an effort to integrate the ideas of [125] (or even the tooling) into the current toolchain.

Lessons learned and going forward. Given the necessary resources, the natural question to ask is whether the benefits justify the means. We believe that, for security-critical code that will be used and replicated in a vast number of applications — which is the case for cryptographic standards — the following take-aways from our work indeed show that the effort is justified:

1. The proof clarifies how the implementation, which has a formal semantics, interprets the specification which is given in paper. Indeed, providing a formal semantics for the specification is an important side result of this line of work, and it allows a complementary effort of verifying the security and correctness of the specification itself.

2. The proof looks at properties that cannot be reasonably checked by testing, such as the fact that the sampling procedures are generating uniform matrices or correctly distributed noise.

3. The proof highlights sub-optimal parts of the code, e.g., the fact that bounds are explicitly stated and proved at the function level permit identifying the points where these bounds are too tight for what is needed,

and hence can be relaxed by removing unnecessary computations. This was indeed the case in this development, where the final version of the AVX2 code is faster because we were able to eliminate redundant reduction steps that became evident when formally proving the ranges of intermediate computations.

As an example of the importance of point (1) above, which is perhaps the less obvious one, let us consider the declarations in Figure 3.10, which pinpoint how SHA3 algorithms are used in Kyber. We note that a total of 9 different types of calls to the SHA3 library are performed: these are declared as operators, which we axiomatize as the semantics of the corresponding functions in the SHA3 library for concrete input and output sizes. We can see that we also annotate these declarations with the security semantics we will assume for these operators in the ongoing follow-up work where we analyze the security of the Kyber specification. These will be either standard PRG/PRF assumptions or, when nothing is stated in the code, they will be modeled as random oracles because the attacker will potentially know the full input. For that future analysis it is important to check that all of these calls to SHA3 algorithms can be considered to be independent hash functions. The argument to justify this claim will rely on the fact that each use of SHA3 algorithms is domain-separated from the other for one of two reasons: 1) using SHAKE vs SHA3 hashing, or 2) if using the same family of algorithms, by using different input and/or output sizes. We do not need to make this argument formal for the purpose of this paper, but it is important to emphasize that our specification is precise enough that we *can* formally state it.

Follow-up work includes tying the functional correctness results we established to formal proofs of correctness for SHA3 implementations, and extending the correctness results to cover other Kyber implementations, namely the faster AVX2 rejection sampling routine and implementations offering protection against timing attacks and Spectre v1 attacks [21]. These results will also be connected to machine-checked security proofs that apply to the Kyber specification we present here.

```
(* PRG that cleans up randomness in PKE *)
op SHA3_512_32_64 : W8.t Array32.t -> unit -> W8.t Array64.t.
clone import HS_DEFS.PseudoRF as HSF with
  type K <- W8.t Array32.t,
  op dK <- srand,
  op F <- SHA3_512_32_64.

(* XOF used to rejection-sample matrix in PKE *)
op SHAKE128_ABSORB_34 : W8.t Array34.t ->  W64.t Array25.t.
op SHAKE128_SQUEEZE_168 : W64.t Array25.t -> W64.t Array25.t *  W8.t Array168.t.
module (XOF : XOF_t) (O : KyberPKE.RO.POracle) = {
  var state : W64.t Array25.t
  proc init(rho : W8.t Array32.t, i j : W8.t) : unit = {
      extseed <- Array34.init
       (fun k => if k < 32 then rho[k] else if k=32 then i else j);
      state <- SHAKE128_ABSORB_34 extseed;
  }
  proc next_bytes() : W8.t Array168.t = {
      (state,buf) <- SHAKE128_SQUEEZE_168 state;
      return buf;
  }
}.

(* PRF used to sample noise in PKE *)
op SHAKE256_33_128 : W8.t Array32.t -> W8.t ->  W8.t Array128.t.
clone import PRF_DEFS.PseudoRF as PRF_ with
  type K <- W8.t Array32.t,
  op dK <- srand,
  op F <- SHAKE256_33_128.

(* PRG that cleans up randomness in KEM *)
op SHA3_256_32_32 : W8.t Array32.t -> unit -> W8.t Array32.t.
clone import HS_KEM_DEFS.PseudoRF as HSF_KEM with
  type K <- W8.t Array32.t,
  op dK <- srand,
  op F <- SHA3_256_32_32.

(* Various Fujisaki-Okamoto hashes, plus KDF used by KEM. *)
op SHA3_256_64_64   : W8.t Array64.t -> W8.t Array64.t.
op SHA3_256_1088_32 : W8.t Array1088.t -> W8.t Array32.t.
op SHA3_256_1184_32 : W8.t Array1184.t -> W8.t Array32.t.
op SHAKE256_64_32 : W8.t Array64.t -> W8.t Array32.t.
module (KemH : KEMHashes) (RO : RO.POracle) = {
  proc pkH(pk : W8.t Array1152.t * W8.t Array32.t) : W8.t Array32.t = {
       return SHA3_256_1184_32 (Array1184.init (fun k => if (k < 1152) then pk.`1[k] else pk.`2[k-1152]));
  }
  proc cH(c : W8.t Array960.t * W8.t Array128.t) : W8.t Array32.t = {
       return SHA3_256_1088_32 (Array1088.init (fun k => if (k < 960) then c.`1[k] else c.`2[k-960]));

  }
  proc g(m : W8.t Array32.t, pkh : W8.t Array32.t) : W8.t Array32.t * W8.t Array32.t  = {
     var ktr;
     ktr <- SHA3_256_64_64 (Array64.init (fun k => if (k < 32) then m[k] else pkh[k-32]));
     return (Array32.init (fun i=> ktr[i]), Array32.init (fun i => ktr[i + 32]));
  }
  proc kdf(kt : W8.t Array32.t, ch : W8.t Array32.t) : W8.t Array32.t = {
       return SHAKE256_64_32 (Array64.init (fun k => if (k < 32) then kt[k] else ch[k-32]));
  }
}.
```

Figure 3.10: SHA3 usage in the KYBER development.

4

Formally verifying Kyber: Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt

This chapter is for all practical purposes identical to the paper *"Formally Verifying Kyber - Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt"* [13], co-authored with José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Peter Schwabe, and Pierre-Yves Strub. Published at CRYPTO 2024.

Abstract. We present a formally verified proof of the correctness and IND-CCA security of ML-KEM, the Kyber-based Key Encapsulation Mechanism (KEM) undergoing standardization by NIST. The proof is machine-checked in EasyCrypt and it includes: 1) A formalization of the correctness (decryption failure probability) and IND-CPA security of the Kyber base public-key encryption scheme, following Bos et al. at Euro S&P 2018; 2) A formalization of the relevant variant of the Fujisaki-Okamoto transform in the Random Oracle Model (ROM), which follows closely (but not exactly) Hofheinz, Hövelmanns and Kiltz at TCC 2017; 3) A proof that the IND-CCA security of the ML-KEM specification and its correctness as a KEM follows from the previous results; 4) Two formally verified implementations of ML-KEM written in Jasmin that are provably constant-time, functionally equivalent to the ML-KEM specification and, for this reason, inherit the provable security guarantees established in the previous points. The top-level theorems give self-contained concrete bounds for the correctness and security of ML-KEM down to (a variant of) Module-LWE. We discuss how they are built modularly by leveraging various EasyCrypt features.

4.1 Introduction

The Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM) is a post-quantum key-encapsulation mechanism that is currently being standardized by NIST as future standard FIPS-203 [167]. ML-KEM is based on Kyber [27, 65] which was selected by NIST in 2022 after round 3 of their multi-year post-quantum cryptography standardization process [3]. Following NIST's announcement, major cryptographic libraries and widely used applications are already transitioning to post-quantum KEMs. Notably, Cloudflare, Google Chrome, and Signal are already using Kyber. It is expected that, once the FIPS-203 standard is ready, these deployments will migrate from Kyber to ML-KEM—for details of the differences between the two see below—and that the number of applications using ML-KEM will rapidly increase.

In this context, it is critically important to ensure that implementations of ML-KEM achieve the targeted security. At a high level, this means that we need to ensure the correctness of security reductions that link the IND-CCA security of Kyber to the module learning with errors (MLWE) problem; we need to ensure that implementations actually match the algorithm specification that has been proven secure; we need to give guarantees about implementation security; and finally we need to make sure that the underlying MLWE problem is indeed hard to solve for the concrete parameters used in ML-KEM.

In this paper we embrace the methods and tools of computer-aided cryptography [35] to give rigorous statements with regards to the first three aspects of this analysis. Concretely, we present a computer-verified proof of IND-CCA security of the ML-KEM draft standard specification down to a variant of the MLWE assumption. We furthermore connect the machine-readable version of the ML-KEM draft specification used in this proof to two efficient implementations in the Jasmin programming language [15, 18]. These implementations are proven functionally correct with respect to the specification and are proven to follow the "constant-time" programming paradigm. Overall, we obtain machine-verified proofs of ML-KEM reaching from the IND-CCA security notion down to the assembly level of highly optimized constant-time software.

The proofs are formalized and machine-checked in the EasyCrypt proof assistant [37]. EasyCrypt is tailored to code-based game-playing cryptographic proofs and supports many common proof techniques, including equivalence up to failure events, lazy sampling, hybrid arguments, and plug and pray. It has been used for proving security of a broad range of cryptographic constructions, and to connect security proofs with implementations (see related work below).

Security Proof. Our main theorem intuitively states the following—the full statement of Theorem 6 is given in Section 4.6.

Theorem 6 (Informal). *The ML-KEM specification is an IND-CCA secure KEM when the key-derivation hash is modeled as a random oracle, under a*

variant of the MLWE assumption where matrices are sampled pseudorandomly using SHAKE-128 and assuming that SHAKE-256 behaves as a pseudorandom function. The failure probability of ML-KEM can be bounded as described in [65].

Our proof consolidates and strengthens prior work in three dimensions: first, our proof is fully machine-checked, and as such provides higher guarantees than pen-and-paper proofs. Second, our proof applies to a (machine-readable) specification of the very recently published ML-KEM draft standard. Third, our proof clarifies the underlying computational assumptions for ML-KEM and streamlines the overall structure of the security analysis. In particular, our work provides a machine-checked proof of the relevant variant of the Fujisaki-Okamoto transform, which we instantiate for the particular case of ML-KEM with concrete parameters. To this end, we lay down and verify all the details related to sampling noise using the non-trivial pseudorandom generation algorithms specified by ML-KEM, under the assumption that various algorithms in the SHA-3 family can be used as pseudorandom functions.

Technically, the security proof is decomposed in two steps. In the first step, we prove security of an abstract ML-KEM specification written in the EasyCrypt proof assistant. This applies to all ML-KEM variants. In the second step, we instantiate our abstract result with a fully concrete specification of the ML-KEM draft standard, with parameters fixed for ML-KEM-768. The final theorem provides a detailed bound for the IND-CCA security of the construction, showing that it is tightly related to a variant of the MLWE problem. This variant of MLWE is tailored to allow plugging in the security proof of the base PKE into the FO transform.[14] In the end our theorem reflects the practical security of ML-KEM-768 closely: MLWE is assumed to be hard when the underlying matrix is sampled exactly as prescribed in the draft standard. For a detailed discussion, see Sections 4.4 and 4.5. Extending these results to other ML-KEM variants is straightforward and will be done when verified implementations are available.

Functional Correctness. It is notoriously hard to develop practical implementations of cryptographic constructions. Subtle discrepancies between specifications, on which security proofs generally reason, and implementations, may have catastrophic security consequences. We provide two Jasmin implementations of ML-KEM-768, and prove that:

> *The reference and vectorized Jasmin ML-KEM implementations are constant-time and functionally equivalent to the ML-KEM specification.*

To obtain this result, we upgrade the two Jasmin implementations of Kyber from [129] to ML-KEM and adapt the proofs of functional correctness to establish

[14] Proving IND-CPA security of the PKE down to standard MLWE is possible assuming that the matrix sampling procedure is a random oracle [122], but this then makes it hard (in a mechanized proof setting) to see the resulting PKE as a deterministic construction that can be plugged into the FO transform.

the link to the EasyCrypt specification of ML-KEM. With this, we provide an end-to-end computer-verified proof that the highly optimized assembly code from [129] (with our adaptations) implements an IND-CCA secure KEM. In order to validate that this KEM is indeed most likely what NIST specified as the draft of ML-KEM, we confirmed that the implementations generate the extensive test vectors recently published by Schmieg from her independent implementation of ML-KEM (see Section 4.7). Finally, to ensure that we follow best practices on mitigating timing side-channel attacks, we used the Jasmin specialized type system to prove that our implementations follow the constant-time discipline.

Relevance. The provable security of Kyber has been scrutinized extensively during the NIST competition. However, the algorithm has undergone several tweaks since the original round-1 submission [190] and the ML-KEM specification has introduced further modifications. Therefore, it is important to consolidate the security proofs during the standardization process. In this work we confirm to a high-level of assurance that the various modifications introduced in Kyber and, more recently in ML-KEM, allow for a proof of security that follows from well-known results published in the literature. We review the evolution of Kyber and the modifications introduced by ML-KEM in Section 4.3.2.

We also contend that connecting security proofs to implementations is necessary to ensure that security guarantees apply to deployed code. Gaps between specifications and implementations can lead to security issues, and arise in practice. For example, early implementations of Kyber computed the "rejection key" used for the implicit rejection in the FO transform, but also returned a non-zero return value in case of rejection, which makes implicit-rejection FO proofs inapplicable. Another example are discrepancies between the specification and implementations of Saber [82] pointed out in [108, Sec. 5.4]. These meant that security proofs in the QROM did not apply to the implemented algorithms.

It is equally important to prove formally that implementations do not leak through timing side channels. For example, it was recently reported by Bhargavan, Kiefer, and Tamvada[15] (and independently by Bernstein), and by Kannwischer and Ravi[16] that some compilers with certain optimization flags translate the reference implementations for Kyber and ML-KEM made available by the Kyber team to binaries containing variable-time DIV instructions operating on secret inputs. Our implementations have been verified to correctly implement the draft standard *and* to avoid secret branch conditions, secret memory addresses, and variable-time DIV and MOD instructions on secret inputs.

[15] See `https://cryspen.com/post/ml-kem-implementation/`.

[16] See `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/ldX0ThYJuBo`.

Limitations. We point out four important limitations of our work. First, our security proof is classical. An important direction for future work is to prove security of ML-KEM against quantum adversaries. This will require using an extension of EasyCrypt for reasoning about security in the Quantum Random Oracle Model [63]. Second, our guarantees are currently limited to the code generated by the Jasmin compiler. In the future, we plan to extend the security and functional correctness guarantees to other implementations using equivalence checking. Third, the complexity bounds of the reductions have been conducted manually. Fourth, our results for failure probability do not go all the way to formally verifying the numbers claimed in [191]. Our proof gives a precise specification of the noise distribution that needs to be computed to obtain such numbers, but refer to [65] for the analytical method to (heuristically) compute this bound. The details are given in Section 4.4.

Related work. There is extensive prior work on security proofs of lattice-based KEMs and related proof techniques (see, e.g., [197, 187, 128, 206, 57, 91, 119, 157]), cryptanalysis of the underlying assumptions (see, e.g., [166, 162, 40, 5, 88, 10, 110, 4, 89]), and side-channel attacks (see, e.g., [181, 111, 113, 176, 165, 115, 116]). However, since the focus and novelty of this paper lie in the field of computer-aided cryptography (CAC), we summarize relevant related work in this field.

One unifying goal of CAC is to deliver machine-checked proofs of security for practical cryptographic implementations. Almeida *et al* [19] provide a general approach to decompose this goal into more classical tasks of functional correctness and side-channel protection of implementation and provable security of specification. This approach has been used successfully for proving indifferentiability from ROM of a Jasmin implementation of SHA3 [17]. Beringer *et al* [44] and Ye *et al* [205] use a similar approach for proving security of HMAC and HMAC-DRBG implementations. In addition, there is a large body of work that establishes either security or functional correctness [35, 59]. Some of these works explicitly target lattice-based constructions. In particular, Hülsing, Meijers and Strub [122] use EasyCrypt to prove CPA security of Saber; Barbosa *et al* [34] use EasyCrypt to prove unforgeability of XMSS; and Barbosa *et al* [33] use EasyCrypt for proving unforgeability of the Dilithium signature scheme. Similarly, Almeida *et al* use EasyCrypt for proving functional correctness of reference and vectorized Jasmin implementations of Kyber [129], and Kreuzer [136] uses the Isabelle proof assistant for proving correctness and IND-CPA security of Kyber.

Most works in CAC establish security against classical adversaries. However, an increasing number of works considers security against quantum adversaries. AutoLWE [39] is an automated tool for proving security of lattice-based cryptographic constructions. It has been used to prove security of several case studies, including Dual Regev encryption [105] and the Micciancio-Peikert encryption scheme [161]. However, constructions such as ML-KEM are outside the scope of AutoLWE. EasyPQC [32] is an extension of EasyCrypt for mechanizing

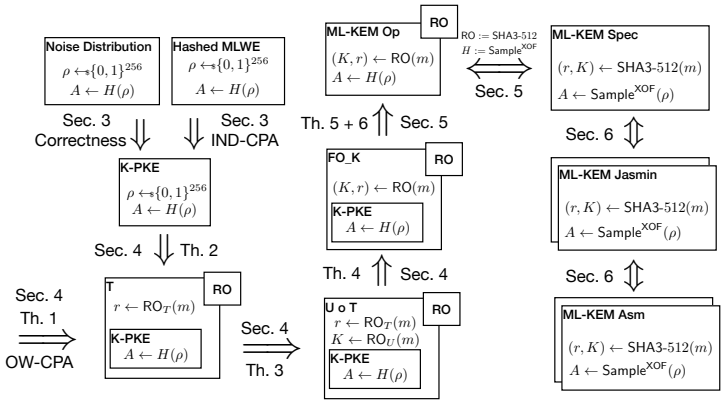


Figure 4.1: Bird's eye view of development. Equivalence symbols represent functional correctness results. Implication symbols represent security and cryptographic correctness (failure probability) proofs.

security proofs against quantum adversaries. EasyPQC supports some (but not all) common reasoning principles for QROM [63], and has been used for proving QROM security of the Full Domain Hash signature and of the Gentry-Peikert-Vaikuntanathan identity-based encryption [105]. Unfortunately, the current version of EasyPQC cannot be used for proving QROM security of Kyber. qrhl-tool is a tool for mechanizing security proofs of classical and quantum cryptographic constructions. The tool, which is implemented on top of the Isabelle proof assistant, is based on a relational Hoare logic for quantum programs [202]. The tool has been used for proving security of the same variant of the Fujisaki-Okamoto transformation we use in this paper [201] in the QROM, but it establishes incomparable bounds to our classical proof. Furthermore, these proofs are not connected to a security proof of a Kyber or ML-KEM specification, nor to an implementation. Finally, recent work, see e.g. [81], studies post-quantum applicability of automated tools for protocol verification. However, these tools cannot be used for verifying ML-KEM.

4.2 Bird's Eye View of Development

Figure 4.1 depicts the global structure of the formally verified development we present throughout the paper. We briefly describe it next.

On the top left, we show the results described in Section 4.4. There we prove that K-PKE, the public-key encryption scheme underlying Kyber and ML-KEM is IND-CPA secure and δ-correct. The IND-CPA proof shows that security holds under a variant of the MLWE assumption, which we call Hashed

MLWE, where the matrix A is sampled using a deterministic procedure H, which is seeded with a uniform random seed ρ. The δ-correctness proof shows that the probability of a decryption failure can be upper-bounded by computing the probability of a noise distribution (independently of any adversarial action) producing a value that exceeds a predefined threshold.

On the right-hand side of the diagram, we show the functional correctness results described in Section 4.7: two Jasmin implementations of ML-KEM-768 are proved to be functionally equivalent to the ML-KEM specification in EasyCrypt. These implementations then give rise to functionally correct and constant-time assembly code via the Jasmin compiler. All of the artifacts on the right-hand side of the diagram are machine-checked representations of the ML-KEM draft standard at different levels of abstraction. In particular, they use SHA3-512 to compute the random coins r and the shared key K required by the FO_K transform. Furthermore, they use a complex sampling procedure $\mathsf{Sample}^{\mathsf{XOF}}$ based on the SHAKE-128 XOF to generate a random-looking matrix from a small seed ρ.

The middle of the diagram shows machine-checked cryptographic proofs for ML-KEM, as described in Sections 4.5 and 4.6. At the top, MLKEM_Op represents the ML-KEM specification with two modifications. The first modification moves the proof to the Random Oracle Model by idealizing the SHA3-512 hash computation for the fixed input size that is used when computing the Fujisaki-Okamoto hash. The second modification is merely syntactic to facilitate the machine-checked proof: we replace the sampling procedure $\mathsf{Sample}^{\mathsf{XOF}}$ with an operator we simply call H. This second modification does not introduce a gap between the ML-KEM specification and the security proof: we prove that the semantics of H is precisely the semantics of $\mathsf{Sample}^{\mathsf{XOF}}$. This allows, in particular, basing our entire proof on the variant of Hashed MLWE that results from fixing H in this way. We guarantee that, indeed, the idealization of SHA3-512 is the only difference between MLKEM_Op and the ML-KEM specification by showing that the two are equivalent when the RO is simply taken to be SHA3-512.

As shown in the middle of the diagram, in Section 4.6 we prove that the IND-CCA security of MLKEM_Op follows from the IND-CCA security of a fully instantiated FO_K transform (Theorems 5 and 6). A full instantiation means plugging in the results described above for K-PKE proved in Section 4.4 and making the algebraic setting concrete for a set of ML-KEM parameters. FO_K uses a single random oracle to obtain random coins r and the session key K, whereas the modularized Fujisaki-Okamoto transform $\mathsf{T} \circ \mathsf{U}^{\not\perp}_m$ uses two independent random oracles to do this. We also prove in Section 4.5, that the IND-CCA security of FO_K follows from the IND-CCA security of $\mathsf{T} \circ \mathsf{U}^{\not\perp}_m$ (Theorem 4). In turn, the security of $\mathsf{T} \circ \mathsf{U}^{\not\perp}_m$ relies on the proof of the T transform (Theorem 2). This theorem shows that one can obtain a OW-secure derandomized public-key encryption scheme starting from an IND-CPA secure one. Finally, as shown on the bottom left of the diagram, we also prove

Theorem 1—not used in the proof of ML-KEM—showing that the T transform also works when applied to a randomized OW-secure public-key encryption scheme.

Artifact. This paper is accompanied by an artifact that permits independently checking the EasyCrypt proofs. The artifact includes a Readme with instructions and the file locations for the EasyCrypt lemmas matching theorems in this paper.

4.3 Preliminaries

We now briefly discuss the mechanized reasoning tools we use for our proofs and give an overview of ML-KEM and how it evolved from Kyber. The cryptographic definitions and notation we use are fairly standard and are given in [14].

4.3.1 The EasyCrypt proof assistant

EasyCrypt[17] [38] is a proof assistant for formalizing proofs of cryptographic properties. Its primary feature is the Probabilistic Relational Hoare Logic (pRHL), which we use throughout to prove equivalences between games, sometimes conditioned by the non-occurrence of a failure event. We call such conditional equivalence steps *up-to-bad* reasoning in later section. pRHL is designed to support reasoning about equivalences of probabilistic programs while reasoning only locally (within oracles) and without reasoning about the distribution of specific variables—essentially keeping track only of the fact that variables in one program are distributed identically to variables in the other, but not keeping track of what that distribution may be. This logic has proved highly expressive for the bulk of cryptographic proof work. However, some steps require more global reasoning (about the entire execution) or keeping track of the distribution of individual variables. Logical rules to support such reasoning steps are implemented in EasyCrypt, but are often unwieldy to apply in concrete context. The EasyCrypt team has, over the years, developed a number of generic libraries that abstract those more complex reasoning rules into "game transformations" or equivalence results that can be instantiated as part of other proofs.

Our proof makes extensive use of three of those generic libraries, which we outline below and discuss more in depth in [14]:

plug-and-pray provides a formalized generic argument for bounding the security loss of reductions that guess an instance, session or query (among a bounded-sized set) will lead to a successful run;

[17] `https://easycrypt.info`

hybrid provides a formalized and generic argument for bounding the distance between two games that differ only in one oracle, but where the transition must be done query-by-query for the purpose of the proof;

PROM provides a generic argument, initially intended to apply to programmable random oracles, that encapsulates the widely used argument that a "value is random and independent of the adversary's view".

4.3.2 Background on ML-KEM

From Kyber to ML-KEM. The original NIST PQC round-1 submission of Kyber described the scheme as a two-stage construction consisting of a module-lattice instantiation of the IND-CPA-secure LPR encryption scheme [152, 150] and a *"slightly tweaked"* Fujisaki-Okamoto (FO) transform [100, 101] to build an IND-CCA-secure KEM. The reduction linking IND-CPA security of the encryption scheme to MLWE was not spelled out in the NIST submission document. For the proof of IND-CCA security of the full scheme the submission mostly referred to previous work on the security of the FO transform, most notably [118] and [186]. Less than two months after the submission documents were made public, D'Anvers pointed out[18] that there are major obstacles to proving IND-CPA security from MLWE in Kyber. The reason is that unlike the original LPR scheme, round-1 Kyber included a public-key-compression step, which invalidates an assumption in the LPR proof. As a consequence, the Kyber submission team decided to tweak the submission for round 2 of NIST PQC by removing the public-key compression (and adjusting some other parameters to obtain similar performance); some more tweaks to parameters were proposed for the round-3 version of Kyber.

Algorithm 7 K-PKE.KeyGen(): key generation

Ensure: Secret key $sk \in \mathcal{R}_q^k$ and public key $pk \in \hat{\mathcal{R}}_q^{\,k} \times \{0,1\}^{256}$
1: $d \leftarrow\!\$ \{0,1\}^{256}$
2: $(\rho, \sigma) \leftarrow \mathrm{G}(d)$
3: $\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho))$
4: $\mathbf{s}, \mathbf{e} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}_\sigma)$ ▷ $\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k$
5: $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
6: $\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
7: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
8: **return** $sk = \hat{\mathbf{s}}$ and $pk = (\hat{\mathbf{t}}, \rho)$

Through the further course of the NIST PQC project, issues were identified also in the second part of Kyber's security argument, which establishes IND-CCA security through a "tweaked" FO transform. It turned out that existing

[18] See `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/PX_Wd11BecI`.

Algorithm 8 K-PKE.$\mathsf{Enc}(pk, m)$: encryption

Require: Public key $pk = (\hat{\mathbf{t}}, \rho) \in \mathcal{R}_q^k \times \{0,1\}^{256}$, message $m \in \{0,1\}^{256}$
Ensure: Ciphertext $c \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$

$r \leftarrow\!\$ \{0,1\}^{256}$
$\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho))$
$\mathbf{r} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}_r)$ ▷ $\mathbf{r} \in \mathcal{R}_q^k$
$\mathbf{e}_1, e_2 \leftarrow \mathsf{CBD}_{\eta_2}(\mathsf{PRF}_r)$ ▷ $\mathbf{e}_1 \in \mathcal{R}_q^k, e_2 \in \mathcal{R}_q$
$\hat{\mathbf{r}} \leftarrow \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{ToPoly}(m)$
$\mathbf{c}_1 \leftarrow \mathsf{Compress}_q(\mathbf{u}, d_u)$
$c_2 \leftarrow \mathsf{Compress}_q(v, d_v)$
return $c = (\mathbf{c}_1, c_2)$

Algorithm 9 K-PKE.$\mathsf{Dec}(sk, c)$: decryption

Require: Secret key $sk = \hat{\mathbf{s}} \in \mathcal{R}_q^k$ and ciphertext $c = (\mathbf{c}_1, c_2) \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$
Ensure: Message $m \in \{0,1\}^{256}$

$\tilde{\mathbf{u}} \leftarrow \mathsf{Decompress}_q(\mathbf{c}_1, d_u)$
$\tilde{v} \leftarrow \mathsf{Decompress}_q(c_2, d_v)$
$m \leftarrow \mathsf{ToMsg}(\tilde{v} - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\tilde{\mathbf{u}})))$
return m

QROM proofs of the FO transform, e.g., from [118], did not apply and that proving the particular variant of the FO transform is not straightforward [108, Sec. 5.4]. Although Kyber's variant of the FO transform was eventually proven secure [157, 30], NIST decided, after discussion on the pqc-forum mailing list, to revert to a more "vanilla" version of the FO transform for ML-KEM.

Our mechanized proof validates the final outcome of these evolutions, which led to the draft specification of ML-KEM. In our models we do *not* include the public-key validation step at the beginning of encapsulation, which NIST introduced in the ML-KEM draft standard. The reason is that, as far as we know, this is the last major open discussion topic for the final standard and it seems rather unlikely that it will be included in the final version of ML-KEM.

Technical description of ML-KEM. We give high-level algorithmic descriptions of K-PKE, the IND-CPA-secure public-key encryption scheme underlying ML-KEM, in Algorithms 7 to 9 and of the full IND-CCA-secure KEM in Algorithms 10 to 12. For a more implementation-oriented description that operates on byte arrays, see [167, Algs. 12–14].

ML-KEM works in the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q = 3329$ and $n = 256$. The core operations are on small-dimension vectors and matrices over $\mathcal{R}_q$; the dimension depends on the parameter k, which is different for different parameter

Algorithm 10 ML-KEM.KeyGen(): key generation

Ensure: Secret key $sk = (sk', pk, \mathrm{H}(pk), z) \in \mathcal{R}_q^k \times (\hat{\mathcal{R}}_q^{\ k} \times \{0,1\}^{256}) \times \{0,1\}^{256} \times \{0,1\}^{256}$ and public key $pk \in \hat{\mathcal{R}}_q^{\ k} \times \{0,1\}^{256}$

1: $(pk, sk') \leftarrow$ K-PKE.KeyGen()
2: $z \leftarrow \{0,1\}^{256}$
3: $sk \leftarrow (sk' \| pk \| \mathrm{H}(pk) \| z)$
4: **return** (sk, pk)

Algorithm 11 ML-KEM.Encap(pk): encapsulation

Require: Public key $pk \in \hat{\mathcal{R}}_q^{\ k} \times \{0,1\}^{256}$
Ensure: Ciphertext $c \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$ and shared key $K \in \{0,1\}^{256}$

1: $m \leftarrow \{0,1\}^{256}$
2: $(K, r) \leftarrow \mathrm{G}(m \| \mathrm{H}(pk))$ ▷ $(K, r) \in \{0,1\}^{256} \times \{0,1\}^{256}$
3: $c \leftarrow$ K-PKE.Enc(pk, m, r)
4: **return** (c, K)

sets of ML-KEM: ML-KEM (NIST security level 1) uses $k = 2$, ML-KEM (NIST security level 3) uses $k = 3$, and ML-KEM (NIST security level 5) uses $k = 4$. We denote elements in $\mathcal{R}_q$ with regular lower-case letters (e.g., v); vectors over $\mathcal{R}_q$ with bold-face lower-case letters (e.g., $\mathbf{u}$), and matrices over $\mathcal{R}_q$ with bold-face upper-case letters (e.g., $\mathbf{A}$).

Algorithm 12 ML-KEM.Decap(c, sk): decapsulation

Require: Ciphertext $c \in \mathcal{R}_{d_u}^k \times \mathcal{R}_{d_v}$ and secret key $sk = (sk', pk, \mathrm{H}(pk), z) \in \mathcal{R}_q^k \times (\hat{\mathcal{R}}_q^{\ k} \times \{0,1\}^{256}) \times \{0,1\}^{256} \times \{0,1\}^{256}$
Ensure: Shared key $K \in \{0,1\}^{256}$

1: $m' \leftarrow$ K-PKE.Dec(sk', c)
2: $(K', r') \leftarrow \mathrm{G}(m' \| \mathrm{H}(pk))$
3: $c' \leftarrow$ K-PKE.Enc(pk, m', r')
4: $K^- \leftarrow \mathrm{J}(z \| c)$
5: **if** $c = c'$ **then then** $K \leftarrow K'$ **else** $K \leftarrow K^-$ ▷ *Requires branch-free conditional*
6: **return** K

In these descriptions, XOF is an extendable output function that in ML-KEM is instantiated with SHAKE-128 [168]. Parse interprets outputs of XOF as sequence of 12-bit unsigned integers and runs rejection sampling to obtain coefficients that look uniformly random modulo q. The function H is instantiated with SHA3-256, G is instantiated with SHA3-512 and J is instantiated with SHAKE256 with output length fixed to 32 bytes. CBD_η denotes sampling

coefficients from a centered binomial distribution with parameter η;[19] extension from coefficients to (vectors of) polynomials is done by sampling each coefficient independently from CBD_η. For example, ML-KEM-768 uses $\eta_1 = \eta_2 = 2$. The sampling routine is parameterized by a pseudorandom function PRF_k with key k. NTT is the number-theoretic transform of a polynomial in $\mathcal{R}_q$. Both input and output of NTT can be written as a sequence of 256 coefficients in $\mathbb{Z}_q$ and typical implementations perform the transform inplace. However, output coefficients do not have any meaning as a polynomial in $\mathcal{R}_q$. We therefore denote the output domain as $\hat{\mathcal{R}}_q$; we apply the same notation for elements in $\hat{\mathcal{R}}_q$, e.g., $\hat{u} = \mathsf{NTT}(u)$. Application of NTT to vectors and matrices over $\mathcal{R}_q$ is done element-wise. $\mathsf{Compress}_q$ compresses elements in $\mathcal{R}_q$ (or $\mathcal{R}_q^k$) by rounding coefficients to a smaller modulus d_v (or d_u). $\mathsf{Decompress}_q$ is an approximate inverse of $\mathsf{Compress}_q$. ToPoly maps 256-bit strings to elements in $\mathcal{R}_q$ by mapping a zero bit to a zero coefficient and mapping a one bit to a $\frac{q}{2}$ coefficient; ToMsg rounds coefficients to bits to recover a message from a noisy version of a polynomial generated by ToPoly.

4.4 K-PKE correctness and IND-CPA security

Our proof of ML-KEM has the same structure as proposed originally for Kyber in [65]. It starts from an IND-CPA public-key encryption scheme and then instantiates a variant of the Fujisaki-Okamoto transform. In this section we look at the underlying public-key encryption scheme, which we call K-PKE as in the ML-KEM draft [167]. We prove IND-CPA security down to a variant of the MLWE assumption; we prove correctness down to the probability of a statistical event that is defined over the randomness space of public-key generation and encryption but, crucially, not the adversary's.

Our presentation in this section uses EasyCrypt notation because the specifications and theorem statements are relatively non-verbose. Later in the paper, we switch to a more concise mathematical notation and point to the artifact for the EasyCrypt formulations. In our presentation we highlight how EasyCrypt allows us to construct the proof modularly. The specification of K-PKE over which we carry out the proof is parametric on all noise distributions, serialization and compression operations, and vector and matrix dimensions. This means it applies to all ML-KEM variants. However, to complete the proof we need a series of hypotheses on these abstract parameters. These are stated as axioms at this level, which means that EasyCrypt will force us to discharge them as proof obligations when we want to use these results for concrete instantiations. Indeed, as we will see in Section 4.6, all the axioms in this section are proved to hold when we apply these general results to the EasyCrypt specification of ML-KEM-768.

[19] This means we have $B(n, p)$ with $p = 1/2$, $n = 2\eta$ and expected value shifted to 0.

4.4.1 The Hashed Module LWE assumption

The computational hardness assumption underlying the security of K-PKE is formalized in our development by first considering an abstract ring R and matrices and vectors over elements in R. We obtain these types, called matrix and vector, and operations over them directly from the Matrix theory that exists in the EasyCrypt library, and then define distributions over them as follows.

```
op [lossless uniform full] duni_R : R distr.
op [lossless] dshort_R : R distr.
op duni = dvector duni_R.
op dshort = dvector dshort_R.
```

Here, the lossless, uniform and full attributes automatically generate axioms that represent assumptions on these distributions. A *full uniform* distribution assigns the same non-zero probability to all values in the support type. In a lossless distribution, the probability of the trivial event true is 1.[20] The dvector operator generalizes a distribution D over ring elements to distributions over (column) vectors by defining distributions that sample each vector element independently from D. Note that we have two distributions over ring elements and vectors thereof: the uniform distribution and an arbitrary distribution that we call *short*. In the variant of the MLWE assumption used in K-PKE this corresponds to sampling ring elements by choosing each polynomial coefficient independently from a centered binomial distribution with $p = 1/2$ and $n = 2$ or $n = 3$, depending on the security parameters.

We define the relevant variant of the MLWE problem, which we call MLWE_H for *Hashed MLWE*, using the experiment in Figure 4.2 (left). There are several differences to the standard MLWE assumption [65]. First and foremost, matrix A is not sampled uniformly at random, but rather created using a deterministic sampling procedure H that expands A from a small seed.

```
type seed.
op H : seed -> matrix.
op [lossless] dseed : seed distr.
```

Sampling A in this way is what K-PKE does in practice and, since our purpose is to connect our security proof to an exact specification of ML-KEM (and from there to an implementation) we cannot go around this aspect as in [136]. Note that, as in [122], we could still prove the security of K-PKE down to the standard notion of MLWE by assuming that H is a random oracle. However, this would create another problem: when using K-PKE to instantiate the FO transform in the next section in a modular way, we must have a version of K-PKE that is *derandomizeable*. Modeling H as a random oracle would make it impossible to plug-in the K-PKE security and correctness results (where H is a random oracle) to the FO formalization (where H must be a deterministic

[20] EasyCrypt's distr type represents subdistributions.

```
module MLWE_H(Adv : HAdv_T) = {
  proc main(tr b : bool) : bool = {
    sd ←$ dseed; s ←$ dshort; e ←$ dshort;
    _A <- if tr then (H sd)^T else (H sd);
    u0 <- _As + e;
    u1 ←$ duni;
    t ←$ duni;
    e' ←$ dshort_R;
    v0 <- t^T s + e';
    v1 ←$ duni_R;
    b' <@ Adv.guess(sd, t, if b
                           then (u1,v1)
                           else (u0,v0));
    return b';
  }

}.
```

```
module MLWE_PKE_HASH_PROC : Scheme = {
  proc kg() : pkey * skey = {
    sd ←$ dseed; s ←$ dshort; e ←$ dshort;
    t <- (H sd)s + e;
    return (pk_encode (sd,t),sk_encode s);
  }

  proc enc(pk : pkey, m : plaintext) : ciphertext = {
    (sd,t) <- pk_decode pk;
    r ←$ dshort; e1 ←$ dshort; e2 ←$ dshort_R;
    u <- (H sd)^T r + e1;
    v <- t^T r + e2 + (m_encode m);
    return c_encode (u,v);
  }

  proc dec(sk : skey, c : ciphertext) : plaintext option = {
    (u,v) <- c_decode c;
    return (Some (m_decode (v − (sk_decode sk^T u))));
  } }.
```

Figure 4.2: Hashed MLWE assumption (left). K-PKE (right).

function). We note that, in practice, the computational assumption underlying ML-KEM is closer to the one we formalize here: one assumes that MLWE holds when A is expanded via a deterministic rejection-sampling procedure based on SHAKE-256.

Another minor difference in our formalization of MLWE_H to the standard MLWE assumption is that, rather than having an MLWE assumption that allows a matrix with a configurable number of rows, our formalization of MLWE fixes A to be square (for some dimensions $k \times k$) and then consider an additional row t^T explicitly. This allows us to have a simpler formalization and reuse our MLWE_H experiment for both the case where we only need to assume that the output vector u is pseudorandom, and the case where we need to assume that both u and an additional ring element v are (jointly) pseudorandom. Finally, because we sample A as H(sd), we need to have two variants of the definition for the case where the challenge is computed over A and A^T.

4.4.2 The K-PKE construction.

K-PKE [65] is a variant of a construction introduced by Lyubashevsky, Peikert, and Regev [152, 151] (LPR). We present our (abstract) specification in Figure 4.2 (right). This construction introduces several new operators: m_encode and m_decode that map plaintexts to ring elements and vice-versa; pk_encode and pk_decode that represent serialization procedures for the public key; sk_encode and sk_decode that represent serialization procedures for the secret key; and, finally, c_encode and c_decode that represent a combination of compression by rounding and serialization for K-PKE ciphertexts. For the security of this construction what these operators do is irrelevant: the IND-CPA security experiment only uses the encoding operators and, if they lose information, this can never hurt

security. However, for correctness, we require additional properties that we will introduce later.

Security of K-PKE. We briefly discuss how we prove that K-PKE is IND-CPA secure under the standard EasyCrypt library definition.[21] This formalizes the standard IND-CPA security notion for PKE. The security proof uses the Hashed MLWE assumption twice to carry out two game hops. The first transformation modifies the key-generation procedure to obtain MLWE_PKE_HASH1 as follows:

```
proc kg() : pkey * skey = {
  sd <$- dseed; s <$- dshort; t <$- duni; return (pk_encode (sd,t), sk_encode s); }
```

In this new game, the t component in the public key is now a uniform vector. We construct an attacker B1 against the Hashed MLWE game that bridges the game hop by running the K-PKE attacker A in a way that perfectly interpolates between the two games. In particular we show:

$$\Pr[\mathsf{CPA[MLWE_PKE_HASH_PROC,A]}() \Rightarrow 1] = \Pr[\mathsf{MLWE_H[B1(A)]}(\mathsf{false},\mathsf{false}) \Rightarrow 1]$$
$$\Pr[\mathsf{CPA[MLWE_PKE_HASH1,A]}() \Rightarrow 1] = \Pr[\mathsf{MLWE_H[B1(A)]}(\mathsf{false},\mathsf{true}) \Rightarrow 1]$$

Here, notation M[·] means the parametrization of module/algorithm M with one or more modules/algorithms. In the proof, B1 attacks Hashed MLWE in the version where the matrix is not transposed (first argument to MLWE_H) and it just uses the first part of the Hashed MLWE challenge in its attack, which is a vector, and simply discards the additional ring element.

The next modification to the game constructs MLWE_PKE_HASH2 where the encryption procedure now simply uses uniformly distributed values to construct the ciphertext as follows:

```
proc enc(pk : pkey, m : plaintext) : ciphertext = {
  _A <- m_transpose (H (pk_decode pk)_1);
  u <$-duni; v <$-duni_R;
  return c_encode (u,v + m_encode m); }
```

Using a similar strategy as before, but reducing to the variant of MLWE_H that uses a transposed matrix, we obtain

$$\Pr[\mathsf{CPA[MLWE_PKE_HASH1,A]}() \Rightarrow 1] = \Pr[\mathsf{MLWE_H[B2(A)]}(\mathsf{true},\mathsf{false}) \Rightarrow 1].$$
$$\Pr[\mathsf{CPA[MLWE_PKE_HASH2,A]}() \Rightarrow 1] = \Pr[\mathsf{MLWE_H[B2(A)]}(\mathsf{true},\mathsf{true}) \Rightarrow 1].$$

Adversary B2 uses the full Hashed MLWE challenge to justify the pseudorandomness of the u and v ciphertext components. Finally, we can use the properties of abstract ring R to show that the ciphertext provided to the adversary is uniformly distributed and therefore independent of m, so that:

$$\Pr[\mathsf{CPA[MLWE_PKE_HASH2,A]}() \Rightarrow 1] = 1/2\,.$$

[21] Line 24 in `https://github.com/EasyCrypt/easycrypt/blob/main/theories/crypto/PKE.ec`

The security result states that, for all A, we have:

$$\begin{aligned}&\Pr[\mathsf{CPA[MLWE_PKE_HASH_PROC,A]}(\,) \Rightarrow 1] - 1/2 = \\ &\quad \Pr[\mathsf{MLWE_H[B1(A)]}(\mathsf{false},\mathsf{false}) \Rightarrow 1] - \Pr[\mathsf{MLWE_H[B1(A)]}(\mathsf{false},\mathsf{true}) \Rightarrow 1] \\ &\quad + \Pr[\mathsf{MLWE_H[B2(A)]}(\mathsf{true},\mathsf{false}) \Rightarrow 1] - \Pr[\mathsf{MLWE_H[B2(A)]}(\mathsf{true},\mathsf{true}) \Rightarrow 1].\end{aligned}$$

Correctness of K-PKE. The correctness experiment we use is the standard definition when analyzing post-quantum secure KEMs using the Fusijaki-Okamoto transform and stated in [118, 117].[22] In this experiment, a correctness adversary gets a freshly challenged *key pair*, including both public and secret key, and tries to find a message m for which the encryption scheme fails to decrypt correctly, i.e., encryption of m followed by decryption does not produce m. The advantage the adversary is given by δ, an upper bound on the failure probability, so that the probability of correct decryption is above $1 - \delta$.

The correctness proof for K-PKE outlined in [65] consists of two steps:

1. Showing that δ can be expressed as the probability that a (complex) noise distribution produces a value above a predefined threshold.
2. Computing the aforementioned probability using an analytic procedure implemented in Python and given in [65].[23]

As noted in [65] and [136], the connection between points (1) and (2) for Kyber and ML-KEM is, so far, heuristic. Indeed, the Python script given in (2) simplifies the analysis of the failure probability by assuming that the distribution of errors introduced by compression can be computed as if the compressed values come from a uniform distribution. This seems to follow from the Hashed MLWE assumption but, in fact and to the best of our knowledge, a formal proof that justifies the heuristic bound has not been given in the literature.[24] Our proof of correctness establishes point (1). We state and prove the following claim for all correctness adversaries A:

$$\begin{aligned}&\Pr[\mathsf{CorrectnessAdv[MLWE_PKE_HASH_PROC,\ A]}(\,) \Rightarrow 1] \leq \\ &\qquad\qquad \Pr[\mathsf{CorrectnessBound}(\,) \Rightarrow 1].\end{aligned}$$

The CorrectnessBound experiment is defined as follows. This experiment has no adversary: it computes a noise value that follows a particular distribution and checks whether this noise value is above a certain bound—concretely, the check is whether any coefficient of the noise polynomial is above $\mathsf{max_noise} - \mathsf{cv_bound_max}$.

[22] As of this work, this is also included in the standard EasyCrypt library: Line 172 in `https://github.com/EasyCrypt/easycrypt/blob/main/theories/crypto/PKE.ec`

[23] The Python script was provided with the Kyber submission to the NIST Post Quantum competition: `https://github.com/pq-crystals/security-estimates/blob/master/Kyber.py`

[24] A reduction to MLWE is possible by expressing the failure probability as a disjunction of events that are simpler to analyze, but this results in significantly worse bounds than the one computed heuristically.

```
module CorrectnessBound = {
  proc main() = {
    sd ←$ dseed; _A <@ H sd; r ←$ dshort; s ←$ dshort;
    e ←$ dshort; e1 ←$ dshort; e2 ←$ dshort_R;
    u  <- m_transpose _Ar + e1;
    cu <- rnd_err_u u;
    n <- e^T r - s^T e1 + e2 - s^T cu.
    return (!under_noise_bound n (max_noise - cv_bound_max));
  } }.
```

So, in short, what our theorem states is that K-PKE is δ-correct, where δ is the probability that the noise distribution, described in experiment CorrectnessBound, produces a value that is above a certain threshold. We explain how we derive this result next. For this, we need to introduce additional definitions and hypotheses. The first requirement is that the serialization operators for public keys and secret keys commute, i.e., that decoding inverts encoding.

```
axiom pk_encodeK : ∀ pk, pk_decode (pk_encode pk) = pk.
axiom sk_encodeK : ∀ sk, sk_decode (sk_encode sk) = sk.
```

Next we need to describe the errors introduced by compression and decompression of the ciphertext as additive noise. We do this by introducing two operators that have precisely this semantics. Note that the semantics of these operators is easy to define when c_encode, c_decode and the ring R are concrete.

```
op rnd_err_v : R -> R.
op rnd_err_u : vector -> vector.

axiom encode_noise u v :
  c_decode (c_encode (u,v)) = (u + rnd_err_u u, v + rnd_err_v v).
```

Finally we need to fix an upper bound for the noise, below which it is guaranteed that the decoded message matches the originally encrypted message.

```
op max_noise : int.
op under_noise_bound : R -> int -> bool.

axiom good_decode m n :
  under_noise_bound n max_noise => m_decode (m_encode m + n) = m.
```

With these operators we can define the full noise expression for K-PKE and restate the correctness experiment in terms of a noise threshold as in Figure 4.3.

In the next step of the correctness proof, we over-approximate the adversary's winning probability by splitting it as the sum of the probability of two (not necessarily disjoint) events. One event failprob2 corresponds to the possibility that the compression error associated with the v component in the ciphertext (cv in the noise expression) is above a certain bound cv_bound. The other event failprob1 is that all the other terms in the noise expression are above max_noise − cv_bound.

```
lemma noise_exp_val _A s e r e1 e2 m :
  noise_exp _A s e r e1 e2 m =
  let t = _As + e in
  let u = _A^T r + e1 in
  let v = t^T r + e2 + (m_encode m) in
  let cu = rnd_err_u u in
  let cv = rnd_err_v v in
    e^T r − s^T e1 − s^T cu + e2 + cv.
```

```
module CorrectnessAdvNoise(A : CORR_ADV) = {
  proc main() = {
    sd ←$ dseed; _A <− H sd;
    r ←$ dshort; s ←$ dshort; e ←$ dshort;
    e1 ←$ dshort; e2 ←$ dshort_R;
    m <@ A.find(pk_encode (sd, _As + e),sk_encode s);
    n <− noise_exp _A s e r e1 e2 m;
    return (!under_noise_bound n max_noise);
  }
}.
```

Figure 4.3: Noise expression (left) and restated correctness experiment (right).

To remove the adversary's influence in the probability of failure, we take the value of cv_bound that guarantees failprob2 = 0. We call this cv_bound_max and note that, in practice, this is much smaller than max_noise (around 10%).

```
op cv_bound_max : int.
axiom cv_bound_valid _A s e r e2 m :
  s∈ dshort => e∈ dshort => r∈ dshort => e2∈ dshort_R =>
  let t = _As + e in
  let v = (t^T r) + e2 + (m_encode m) in
    under_noise_bound (rnd_err_v v) cv_bound_max.
```

At this point we can move to a correctness game where the probability of failure does not depend on the adversary and is simply failprob1. This game is CorrectnessBound that we use in our theorem statement. To summarize, we formally verify that K-PKE is δ-correct for δ = failprob1.

We finally note that by "maxing-out" the adversary's contribution to the noise, we do not need to consider the distribution of cv at all. However, the distribution of cu to the noise expression computed in game CorrectnessBound still does not match the one in the heuristic computation proposed in [65], where cu results from compressing a uniform u. Justifying this simplification using (Hashed) MLWE is not immediate because r and e_1 occur elsewhere in the noise expression, so an efficient reduction that does not get r and e_1 cannot check for the failure event.

4.4.3 Derandomizing K-PKE

Looking at the K-PKE specification in Section 4.3, we see that the key generation and encryption algorithms take small random seeds as inputs and then use complex procedures to convert these seeds into the required noise values. Note also that these procedures internally use hash functions that we will later model as pseudorandom functions to expand the input seeds into longer sequences of pseudorandom bits. To bridge this gap to the abstract specification of K-PKE that we introduced in this section, and also to allow connecting K-PKE to the Fujisaki-Okamoto transform, we specify an alternative definition of K-PKE that

```
type randomness.
op [uniform full lossless]drand :
                    randomness distr.
op prg_kg : randomness ->
                    seed * vector * vector.
op prg_enc : randomness ->
                    vector * vector * R.

op kg(r : randomness) : pkey * skey =
  let (sd,s,e) = prg_kg r in
  let t =  (H sd)s + e in
    (pk_encode (sd,t),sk_encode s).

op enc(rr : randomness, pk : pkey, m : plaintext) : ciphertext =
  let (sd,t) = pk_decode pk in
  let (r,e1,e2) = prg_enc rr in
  let u = m_transpose (H sd)r + e1 in
  let v = t^T r) + e2 + (m_encode m) in
    c_encode (u,v).

op dec(sk : skey, c : ciphertext) : plaintext option =
  let (u,v) = c_decode c in
    Some (m_decode (v - (sk_decode sk)^T u)).
```

Figure 4.4: De-randomized K-PKE.

explicitly uses abstract deterministic procedures to expand small seeds of type randomness into the required noise values. This is shown in Figure 4.4.

It is not hard to see that the correctness and security of derandomized K-PKE follow—and we prove this in EasyCrypt—from the results we proved earlier in this section, under the assumption that prg_kg and prg_enc produce distributions that are computationally close to the following ideal randomness distributions.

```
op prg_kg_ideal = dseed `*` dshort `*` dshort.
op prg_enc_ideal = dshort `*` dshort `*` dshort_R.
```

Where d_1 `*` d_2 represents the product of the distributions d_1 and d_2.

In the next section we discuss how we formalized the Fujisaki-Okamoto transform used in ML-KEM. When we instantiate it, we will use the derandomized K-PKE we just defined. Then, in Section 4.6, when we apply the result to a fully concrete specification of K-PKE, we prove that the procedures in ML-KEM that instantiate prg_kg and prg_enc are indeed producing distributions computationally close to the ideal ones, under the assumption that the underlying hash functions have standard pseudorandomness properties. This will allow us to have a bound for ML-KEM expressed as a function of the Hashed MLWE assumption, the δ-correctness of K-PKE as defined in the previous section, and the pseudorandomness of algorithms in the SHA3 family.

4.5 Formalizing the FO Transform used in ML-KEM

Hofheinz, Hövelmanns and Kiltz [118, 117] (HHK) give a modular treatment of the Fujisaki-Okamoto (FO) transform [100, 101] that splits it into two constructions presented and analyzed independently: the T and the U transforms.

The T transform (Fig. 4.5) constructs an actively OW-secure *deterministic* PKE scheme (active in the sense that the adversary is given access to both a ciphertext validity oracle and to a plaintext checking oracle [14]), from a

passively secure *probabilistic* PKE scheme (one can start from IND-CPA or OW-CPA). HHK [117, Theorems 3.1 and 3.2] bound the insecurity of this construction.

The U transform (Fig. 4.6) constructs an IND-CCA-secure KEM from a OW-secure PKE scheme (for example, such as that constructed by T). It operates by encrypting (using the PKE scheme), a random message m which is treated as a pre-key from which a shared key k is derived through a random oracle. HHK describe four variants of U: $\mathsf{U}^{\perp}$, $\mathsf{U}^{\not\perp}$, $\mathsf{U}^{\perp}_{m}$ and $\mathsf{U}^{\not\perp}_{m}$. The superscript denotes whether rejection when the underlying PKE fails to recover a pre-key is explicit ($\perp$), or implicit ($\not\perp$) (that is, it returns a random-looking shared key instead of an error). The subscript indicates whether the key derivation step includes only the pre-key m, or also the ciphertext (no subscript). HHK [117, Theorems 3.3-3.6] demonstrate the IND-CCA security of the KEM resulting from the different versions of the U transform, with varying one-wayness assumptions on the underlying PKE based on the U variant used. For example, it is convenient to assume one-wayness in the presence of a ciphertext validity oracle when the KEM does explicit rejection.

Each of the transforms uses a random oracle. Following HHK, *in this section*, we call these random oracles G (used by T) and H (used by U). These are distinct from the oracles H and G used in other sections, and in particular in the description of ML-KEM in Section 4.3.

Overview of our results on FO.

ML-KEM can be seen as an instance of the composed transform $\mathsf{U}^{\not\perp}_{m} \circ \mathsf{T}$. However, we do not follow the purely modular proof structure introduced by HHK [118, 117]. As pointed out in various works [47, 120], the abstraction boundary introduced between the T and U transforms in [118, 117] makes the proof of the U transform unnecessarily complex. For example, the U transform requires a property called *rigidity*[25] from the underlying PKE, which is guaranteed by the T transform by construction.

In our formalization we consider the $\mathsf{U}^{\not\perp}_{m}$ transform specifically in the case where it is composed with the T transform. This eases some proof steps, in particular when query counting must across the abstraction boundary.

4.5.1 The T transform

Let $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a derandomized (as in Section 4.4) public-key encryption scheme with message space $\mathcal{M}$ and randomness space $\mathcal{R}$, and $\mathsf{G} : \mathcal{M} \to \mathcal{R}$ be a random oracle. The T transform defines a new public-key encryption scheme $\mathsf{PKE}_1 = \mathsf{T}[\mathsf{PKE}, \mathsf{G}] = (\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$, where algorithms Gen_1, Enc_1 and Dec_1 are shown in Figure 4.5.

[25] The PKE must guarantee consistent re-encryption unless decryption outputs $\perp$.

$\underline{\mathsf{Enc}_1(pk, m)}$:	$\underline{\mathsf{Dec}_1((pk, sk), c)}$:
1: $c \leftarrow \mathsf{Enc}(pk, m; \mathsf{G}(m))$	1: $m' \leftarrow \mathsf{Dec}(sk, c)$
2: **return** c	2: **if** $m' = \perp$ **or** $\mathsf{Enc}(pk, m'; \mathsf{G}(m')) \neq c$ **then**
	3: **return** $\perp$
	4: **else return** m'

Figure 4.5: The T transform: algorithms for $\mathsf{T}[\mathsf{PKE}, \mathsf{G}]$ for $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Gen_1 runs Gen and sets the secret key to be (pk, sk).

Our formal verification of the ML-KEM IND-CCA security proof includes a machine-checked proof of the following security and correctness theorem for the T transform.

Theorem 1. *Assume* PKE *to be* δ*-correct and* γ*-spread. Then* $\mathsf{PKE}_1 = \mathsf{T}[\mathsf{PKE}, \mathsf{G}]$ *is* $(q_G + 1) \cdot \delta$*-correct, where* q_G *is the number of random oracle queries placed by the correctness attacker. Furthermore, for any* OW-PCVA *adversary* $\mathcal{B}$ *that issues at most* q_{G} *queries to the random oracle* G*,* q_P *queries to a plaintext checking oracle* Pco*, and* q_V *queries to a validity checking oracle* Cvo*, there exists a* OW-CPA *adversary* $\mathcal{A}$ *whose running time is about that of* $\mathcal{B}$*, and such that*

$$\mathsf{Adv}^{\mathsf{OW-PCVA}}_{\mathsf{PKE}_1}(\mathcal{B}) \leq (q_{\mathsf{G}} + q_P + 2) \cdot \delta + q_V \cdot 2^{-\gamma} + (q_{\mathsf{G}} + q_P + 1) \cdot \mathsf{Adv}^{\mathsf{OW-CPA}}_{\mathsf{PKE}}(\mathcal{A})$$

The theorem statement differs from that given in HHK [118, 117, Theorem 3.1] only in that there is an extra additive δ term in the correctness bound, and an extra $2 \cdot \delta$ additive term in the security bound. In both cases, the high-level proof is identical to those given by HHK, except for the fact that our δ-correctness reductions make one additional call to the random oracle before terminating, to ensure that a specific message is in the random oracle table. This simplifies some intermediate steps in the proof. We sketch the proof next, highlighting the steps that were more challenging to formalize in EasyCrypt.

Formalization Sketch. The correctness proof, which takes a paragraph to write [118, 117], is surprisingly involved in EasyCrypt. Its intuition is easy to understand: if the adversary finds a correctness error for the T transform, then there must be some message in the random oracle table for which the underlying PKE has failed. Writing this in game-based form requires: 1) wrapping the correctness adversary to ensure the problematic message is in the random oracle table; 2) guessing which of the adversary's queries to the random oracle will be problematic; 3) proving that, if a correct guess is made, the probability of error in T is the same as in the base PKE. Step 3) is particularly laborious in EasyCrypt: it entails moving a randomness sampling that occurs due to an adversarial call to the random oracle, in order to place it at the beginning of

the game. This aligns the problematic sampling with the single randomness sampling that occurs in the correctness game for PKE. An instance of the PROM library [14] forms the core of the argument, but remains difficult to instantiate since we cannot predict the inputs the adversary will query the random oracle on. Step (2) is performed using the generic *plug-and-pray* argument [14].

The security proof generally follows along the steps outlined by HHK [118, 117]. The first game hop (Game 1) changes the ciphertext validity oracle to reject all ciphertexts that cannot be reconstructed using a prior query to G. This hop is summarized in the handwritten proof as following from a union bound over all queries to the validity oracle. In EasyCrypt this must be expressed as a rather intricate hybrid argument that handles one query at a time. This is formalized using the existing generic *hybrid* argument.

The next step (Game 2) removes the use of the secret key from both the ciphertext validity and the plaintext checking oracles. Both checks are replaced by re-encryption; an inconsistency can occur only if the correctness of the T transform itself fails. This hop is formalized as a simple *up-to-bad* argument, followed by (tightly) bounding the probability of the failure event—a failure in the correctness of T—using the existing result on T correctness. This reduction places at most $(q_{\mathsf{G}} + q_P + 1)$ queries to its random oracle, which justifies the δ-correctness term in the security theorem statement.

The final game (Game 3) is reached by declaring the adversary a loser if she queries the OW challenge message to G—an event we call QUERY. Since the secret key is not used in any of the oracles at this point, the probability of success in the final game follows from a simple reduction from the one-wayness of the underlying PKE. Formally bounding the probability of the QUERY event is harder, since it once again involves guessing which random-oracle query made by the adversary will actually break the one-wayness of the underlying PKE. This is resolved using a similar argument as that deployed in the correctness proof.

Combined, the two reductions from one-wayness yield the $(q_{\mathsf{G}} + q_P + 1) \cdot \mathsf{Adv}^{\mathsf{OW\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A})$ term in the theorem, where $(q_{\mathsf{G}} + q_P)$ is the guessing loss. ■

Theorem 1 does not directly apply to the ML-KEM construction, which starts from an IND-CPA —rather than OW-CPA —encryption scheme. For this reason, we extend the proof of Theorem 1 to obtain the analogous of [118, 117, Theorem 3.2], as follows.

Theorem 2. *Assume* PKE *to be δ-correct and γ-spread. Then, for any* OW-PCVA *adversary $\mathcal{B}$ that issues at most q_{G} queries to the random oracle* G*, q_P queries to a plaintext checking oracle* Pco*, and q_V queries to a validity checking oracle* Cvo*, there exists an* IND-CPA *adversary $\mathcal{A}$ whose running time is about that of $\mathcal{B}$ and such that*

$$\mathsf{Adv}^{\mathsf{OW-PCVA}}_{\mathsf{PKE}_1}(\mathcal{B}) \leq (q_{\mathsf{G}} + q_P + 3) \cdot \delta + q_V \cdot 2^{-\gamma} + \frac{2(q_{\mathsf{G}} + q_P + 1)}{|\mathcal{M}|} + 4 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A})$$

$\mathsf{Gen}_2()$:	$\mathsf{Enc}_2(pk)$:	$\mathsf{Dec}_2((sk, s), c)$:
1: $s \leftarrow\!\!\$\ \mathcal{K}$	1: $m \leftarrow\!\!\$\ \mathcal{M}$	1: $m' \leftarrow \mathsf{Dec}_1^{\mathsf{G}}(sk, c)$
2: $(pk, sk) \leftarrow\!\!\$\ \mathsf{Gen}_1()$	2: $c \leftarrow \mathsf{Enc}_1^{\mathsf{G}}(pk, m)$	2: **if** $m' \neq \perp$ **then**
3: **return** $(pk, (sk, s))$	3: $k \leftarrow \mathsf{H}(m)$	3: **return** $\mathsf{H}(m')$
	4: **return** (k, c)	4: **else return** $\mathsf{J}(s, c)$

Figure 4.6: The $\mathsf{U}_m^{\not\perp}$ transform.

Sketch. There are two differences to the proof of Theorem 1, where we replace the reductions from OW-CPA with reductions from IND-CPA. However, as discussed by HHK [118, 117], here we can take advantage of the fact that IND-CPA is a stronger property than OW-CPA, in which the attacker controls almost fully the encrypted message, to obtain a tighter reduction. In particular, IND-CPA security implies a multi-guess version of one-wayness where the OW-CPA adversary outputs a list of guesses. This is proved generically in EasyCrypt, as discussed in [14]. It enables us to more tightly bound the probability of QUERY, constructing a OW adversary that outputs the full contents of the random oracle table rather than trying to guess which of its entries is the correct one. Our formalization is structured so that the core arguments in the proofs of Theorem 1 and 2 are proven once and used twice, rather than repeated. ■

Remark. We use the reduction from IND-CPA to multi-guess OW-CPA described above twice to derive Theorem 2. The most intuitive use is in tightly bounding QUERY, as we described above. However, we also use it to convert the OW adversary constructed for Game 3 in the proof of Theorem 1 into an IND-CPA adversary in a black-box way by making it return a list of size 1. This second use justifies the minor difference to the bound given by HHK [118, 117].

4.5.2 The $\mathsf{U}_m^{\not\perp}$ transform.

Let $\mathsf{PKE}_1 = (\mathsf{Gen}, \mathsf{Enc}_1, \mathsf{Dec}_1)$ be a deterministic PKE scheme that relies on a random oracle G. The $\mathsf{U}_m^{\not\perp}$ transform uses a new random oracle $\mathsf{H} : \mathcal{M} \to \{0,1\}^n$ and a pseudorandom function $\mathsf{J} : \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$, where $\mathcal{K}$ is the key space, to define a new key encapsulation mechanism $\mathsf{KEM}_m^{\not\perp} = \mathsf{U}_m^{\not\perp}[\mathsf{PKE}_1, \mathsf{H}] = (\mathsf{Gen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ whose algorithms are shown in Figure 4.6. We note an important difference to the version of U given by HHK [118, 117]: we use a PRF primitive rather than a random oracle to generate (random-looking) shared keys when performing implicit rejection in decapsulation. This simplifies the proof and implies that our bounds include PRF distinguishing advantage terms rather than concrete probability terms to account for the possibility that the adversary exploits a weakness in J in its attack.

```
proc kg() : pkey * skey = {
  (pk,sk) <$- kg;
  k <$- dK;
  return (pk, ((pk,sk),k));
}
```

```
proc enc(pk : pkey)
    : ciphertext * key = {
  m <$- dplaintext;
  c <@TT(HT).enc(pk,m);
  k <@ HU.get(m);
  return (c,k); }
```

```
proc dec(sk : skey, c : ciphertext)
                : key option = {
  m' <@ TT(HT).dec(sk[1],c);
  if (m' = None) { k <- J sk[2] c; }
  else { k <@ HU.get(oget m'); }
  return (Some k); }
```

Figure 4.7: EasyCrypt specification of $\mathsf{KEM}_m^{\not\perp} = \mathsf{U}_m^{\not\perp}[\mathsf{PKE}_1, \mathsf{H}]$.

Also note that, if one does not fix a concrete instance for PKE_1, then one must treat the interaction with G generically in the proof. For example, in [118, 117, Theorem 3.6], the bound is given as a function of the maximum number of queries placed by Enc_1 and Dec_1 to G. This reasoning is possible in EasyCrypt, but it would significantly complicate the proof. For this reason, we consider explicitly the case where $\mathsf{PKE}_1 = \mathsf{T}[\mathsf{PKE}, \mathsf{G}]$. The EasyCrypt specification of this composed transform is given in Figure 4.7. We prove the following result in EasyCrypt, which is the analogue of HHK's Theorem 3.6 [118, 117] in this more concrete setting.

Theorem 3. *If* $\mathsf{PKE}_1 = \mathsf{T}[\mathsf{PKE}, \mathsf{G}]$ *is* $\delta_1(\cdot)$*-correct, where* δ_1 *is parameterized by the number of queries of a correctness adversary to* G*, then* $\mathsf{KEM}_m^{\not\perp}$ *is* $\delta_1(0)$ *correct. Furthermore, for any* IND-CCA *adversary* $\mathcal{B}$ *against* $\mathsf{KEM}_m^{\not\perp}$*, issuing at most* q_D *queries to the decapsulation oracle, and at most* q_G*, and* q_H *queries to its random oracles* G *and* H *respectively, there exist a* OW-CPA *adversary* $\mathcal{A}$ *against* PKE_1 *and a* PRF *adversary* $\mathcal{D}$ *against* J *whose running times are about that of* $\mathcal{B}$ *and such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}_m^{\not\perp}}(\mathcal{B}) \leq \mathsf{Adv}^{\mathsf{OW\text{-}CPA}}_{\mathsf{PKE}_1}(\mathcal{A}) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{J}}(\mathcal{D}) + 2 \cdot \delta_1(q_\mathsf{G} + q_H + 2)$$

Adversary $\mathcal{A}$ *makes at most* $q_\mathsf{H} + q_\mathsf{G} + 1$ *queries to* G.

Since the proof of Theorem 3.6 is not given in detail by HHK [118, 117], we provide the full details. The proof is close, at a high level, to that of Theorem 3.5 [118, 117], with simplifications that come from our specific setting.

Proof. The correctness proof is straightforward. The KEM correctness experiment has no adversary, which means that we can construct an adversary against the correctness of PKE_1 by simply sampling a random message and outputting it. The resulting PKE_1 correctness game is identical to the KEM correctness game. Correctness follows from the fact that this PKE_1 adversary makes no calls to G.

For the security proof, we construct a sequence of games. The first hop (Game 1) simply replaces J with a random function, that is outside of the adversary's view, with the same name $\mathsf{J} : \mathcal{M} \to \{0,1\}^n$. The proof of this hop is a direct reduction ($\mathcal{D}$ in Theorem 3) from the PRF property of J.

Game 1

Decaps(c):

1: If $c = c^*$ **return** $\perp$
2: $m' \leftarrow \mathsf{Dec}(sk, c)$
3: **if** $m' = \perp$ **or** $\mathsf{Enc}(pk, m'; \mathsf{G}(m')) \neq c$ **then**
4: **return** $\mathsf{J}(c)$
5: **else return** $\mathsf{H}(m')$

H(m):

1: **if** $m \in T_{\mathsf{H}}$ **then return** $T_{\mathsf{H}}[m]$
2: $k \leftarrow\!\!\$ \{0,1\}^n$
3: $T_{\mathsf{H}}[m] \leftarrow k$
4: **return** k

Game 2

Decaps(c):

1: If $c = c^*$ **return** $\perp$
2: If $c \in \mathcal{L}_D$ **return** $L_D[c]$
3: $k \leftarrow\!\!\$ \{0,1\}^n$
4: $\mathcal{L}_D[c] \leftarrow k$
5: **return** k

H(m):

1: **if** $m \in T_{\mathsf{H}}$ **then return** $T_{\mathsf{H}}[m]$
2: $c \leftarrow \mathsf{Enc}(pk, m; \mathsf{G}(m))$
3: **if** $\mathsf{Dec}(sk, c) = \perp$ **or** $\mathsf{Dec}(sk, c) \neq m$ **then**
4: $\mathsf{bad}_2 \leftarrow 1$
5: **if** $m = m^*$ **and** $\mathsf{Dec}(sk, c^*) = m^*$ **then**
6: $\mathsf{bad}_3 \leftarrow 1$
7: **if** $c \in \mathcal{L}_D$ **then then** $k \leftarrow \mathcal{L}_D[c]$
8: **else** $k \leftarrow\!\!\$ \{0,1\}^n$; $\mathcal{L}_D[c] \leftarrow k$
9: $T_{\mathsf{H}}[m] \leftarrow k$
10: **return** k

Figure 4.8: Game 1 to Game 2 hop. In blue: bad event for Game 3.

The jump to Game 2 modifies the decapsulation oracle to rely only on a table $\mathcal{L}_D$. If the input ciphertext c is not in that table, a fresh shared key k is sampled and returned to the adversary; $\mathcal{L}_D$ is updated to ensure consistent simulation of later queries. We also modify random oracle H to guarantee consistency with $\mathcal{L}_D$: whenever a fresh message m is queried to H, we check whether its encryption under PKE_1 is in $\mathcal{L}_D$. If so, then we program H consistently with the corresponding entry in $\mathcal{L}_D$. Otherwise, we lazily sampling a fresh key k, and update *both* $\mathcal{L}_D$ and H. We also set a bad_2 flag in oracle H whenever a correctness error occurs in PKE that would lead to an inconsistency in $\mathcal{L}_D$, and declare the adversary a loser if bad_2 ever occurs. Figure 4.8 shows pseudocode for Games 1 and 2.

Interestingly, we can show that the probability of bad_2 can be bounded by the correctness of PKE_1, which allows us to avoid performing once again an intricate reduction that involves dealing with random oracle G. Our reduction places at most $q_{\mathsf{H}} + q_{\mathsf{G}} + 2$ queries to its G oracle: 2 queries from the KEM challenge, q_{H} queries to simulate the H oracle as in Game 2, and q_{G} queries to forward $\mathcal{A}$'s G queries. This hop accounts for one of the δ_1 terms in Theorem 3.

It is less straightforward to prove that the two games are, in fact, identical until bad_2 occurs. It implies proving that there are only two possible cases for entries in $\mathcal{L}_D$ created during a call to the decapsulation oracle:

- Those that produce keys that would correspond to implicit rejections;
- Those that would lead to the decapsulation oracle calling H to return a valid shared key, in which case the patching done inside H leads to an adversarial view that is identical to the previous game.

We prove this step by working over a version of the games where random oracle G is eagerly sampled as a random function that is fixed throughout both games—this is done by instantiating the PROM library [14]. Eagerly sampling random oracle outputs is important in the proof, as it allows us to consider PKE_1 as a fully deterministic scheme (rather than a scheme that merely appears deterministic), which is one of the requirements of HHK's Theorem 3.6 [118, 117]. It is also very convenient from a formal verification point of view, as it permits writing an intuitive, yet non-trivial, invariant that keeps track of the case analysis above. (See Figure 4.9.) Note that this invariant only needs to hold until bad_2 occurs. We distinguish the variables in Games 1 and 2 by denoting them with $\{1\}$ and $\{2\}$, respectively, and capture with $\mathsf{goodc}(c, sk, \mathsf{G})$ the fact that a ciphertext is consistent under re-encryption (as defined by sk and G). Here is a breakdown of the invariant point by point:

- Equations 4.1 and 4.2: There is a consistency between invalid ciphertexts in $\mathcal{L}_D$ in Game 2 and the values stored in J in Game 1.
- Equations 4.3, 4.6 and 4.7: There is a consistency between valid ciphertexts in $\mathcal{L}_D$ in Game 2 and the values stored in H in Game 1, which may include ciphertexts not yet synchronized with H_2 in Game 2.
- Equations 4.4, 4.5 and 4.6: H_2 in Game 2 guarantees consistency with H_1 and Decaps in Game 1 via $\mathcal{L}_D$.

Game 3 is modified to create the challenge encapsulation without calling H, sampling the shared key uniformly at random. In this game the adversary has no advantage in distinguishing the challenge encapsulation key from a random one. This claim is relatively straightforward to verify in EasyCrypt using probabilistic Hoare logic. It remains to bound the distance between Games 2 and 3.

Again, this is done by an up-to-bad argument: the two games are identical unless H is queried on the message $m^\star$ encrypted under PKE_1 in the challenge encapsulation. This is shown as bad_3 in Figure 4.8, and we note that we actually capture a more fine-grained event that is closer to the winning condition of the $\mathsf{OW\text{-}CPA}$ game for PKE_1. Proving that the two games are identical until bad_3 occurs is straightforward using the logics of EasyCrypt, and relying on the fact that the adversary can only win at this point if no correctness error has occurred due to the action of bad_2.

$$\forall c, \quad c \in l_D\{2\} \wedge \neg \mathsf{goodc}(c, sk, \mathsf{G}) \Rightarrow c \in \mathsf{J}\{1\} \tag{4.1}$$
$$\wedge\ \forall c, \quad c \in \mathsf{J}\{1\} \Rightarrow \mathcal{L}_D\{2\}[c] = \mathsf{J}\{1\}[c] \tag{4.2}$$
$$\wedge\ \forall c, \quad c \in \mathcal{L}_D\{2\} \wedge \mathsf{goodc}(c, sk, \mathsf{G}) \Rightarrow \mathsf{Dec}_1(sk, c) \in \mathsf{H}\{1\} \tag{4.3}$$
$$\wedge\ \forall m, \quad m \in \mathsf{H}\{2\} \Rightarrow \mathsf{Enc}(m, pk, \mathsf{G}(m)) \in \mathcal{L}_D\{2\} \tag{4.4}$$
$$\wedge\ \mathcal{L}_D\{2\}[\mathsf{Enc}(m, pk, \mathsf{G}(m))] = \mathsf{H}\{2\}(m) \tag{4.5}$$
$$\wedge\ \mathsf{H}\{1\}(m) = \mathsf{H}\{2\}(m) \tag{4.6}$$
$$\wedge\ \forall m, \quad m \in \mathsf{H}\{1\} \wedge m \notin \mathsf{H}\{2\} \Rightarrow \mathcal{L}_D\{2\}[\mathsf{Enc}(m, pk, \mathsf{G}(m))] = \mathsf{H}\{1\}(m) \tag{4.7}$$

Figure 4.9: Invariant for Game 1 to Game 2 hop in the proof of Theorem 3.

It remains to bound the probability of bad_3 occurring, using a reduction from the OW security of PKE_1. The reduction, of course, cannot directly check whether bad_3 has occurred using the secret key, and instead simply checks whether a solution to the OW challenge is found by checking that $\mathsf{Enc}(pk, m; \mathsf{G}(m)) = c^*$. There is therefore a small gap between the probability of the reduction winning the one-wayness game and the occurrence of bad_3: this occurs if the challenge ciphertext decrypts to something other than m^*. We carry out a final up-to-bad analysis of our reduction, and show that it does break the one-wayness of the underlying scheme, except if there is a correctness error in PKE_1. This accounts for the remaining δ_1 terms in Theorem 3.

■

4.5.3 The FO transform in ML-KEM

The FO transform used in ML-KEM, FO_K, is shown in Figure 4.10. It uses a *single* random oracle GH, but it is identical to the $\mathsf{KEM}^{\not\perp}_m$ transform considered above if one sees GH as the product of two independent random oracles $\mathsf{G} \times \mathsf{H}$. Using exactly this intuition, we prove the following theorem in EasyCrypt.

$\mathsf{Gen}_K()$:	$\mathsf{Enc}_K(pk)$:	$\mathsf{Dec}_K((sk, s), c)$:
1: $s \leftarrow_{\$} \mathcal{K}$	1: $m \leftarrow_{\$} \mathcal{M}$	1: $m' \leftarrow \mathsf{Dec}(sk, c)$
2: $(pk, sk) \leftarrow$ $_{\$}\,\mathsf{Gen}()$	2: $(r, k) \leftarrow$ $\mathsf{GH}(m)$	2: $(r, k) \leftarrow \mathsf{GH}(m')$
3: **return** $(pk, (sk, s))$	3: $c \leftarrow$ $\mathsf{Enc}(pk, m; r)$	3: $k' \leftarrow \mathsf{J}(s, c)$
	4: **return** (k, c)	4: **if** $m' = \perp$ **or** $\mathsf{Enc}(pk, m'; r) \neq c$ **then**
		5: **return** k'
		6: **else return** k

Figure 4.10: The FO transform used in ML-KEM, denoted FO_K.

Theorem 4. *Assume* PKE *to be δ-correct. Then* FO_K *is δ-correct. Furthermore, for any* IND-CCA *adversary $\mathcal{B}$ against* FO_K*, issuing at most q_D queries to the decapsulation oracle, and at most q_{GH} queries to its random oracle* GH*, there exist an* IND-CPA *adversary $\mathcal{A}$ against* PKE *and a* PRF *adversary $\mathcal{D}$ against* J *s.t.*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{FO}_K}(\mathcal{B}) \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{J}}(\mathcal{D}) + Q \cdot \delta + \frac{2(2q_{\mathsf{GH}}+2)}{|\mathcal{M}|} + 4 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A})$$

for $Q = 3 \cdot (2q_{\mathsf{GH}} + 3) + 1$ and the running times of $\mathcal{A}$ and $\mathcal{D}$ are about that of $\mathcal{B}$.

Sketch. Both the correctness and the security statements follow from the correctness and security of KEM^{ι}_{m} by direct reductions. These reductions perfectly simulate calls to GH by calling both G and H. A simple invariant in EasyCrypt then permits showing that all values that $\mathcal{B}$ has seen in GH exactly match those queried by the reduction to G and H. Furthermore, this invariant is preserved on every fresh call by arguing that a sample at the output (r, k) of GH of type $\mathcal{R} \times \mathcal{K}$ is identically distributed to the value that results from grouping (r, k) coming from independent (guaranteed to be fresh) calls to G and H. Using this invariant, it is straightforward that the reductions perfectly simulate FO_K while attacking KEM^{ι}_{m} with the same advantage as $\mathcal{B}$. Then, the theorem follows from instantiating Theorem 3 with the bound in Theorem 2. ■

4.6 Main result: Security of ML-KEM

The main theorem in our machine-checked proof of ML-KEM applies to an EasyCrypt specification of the ML-KEM draft standard [167], obtained with minor and necessary adaptations from the KYBER specification used in [129]. This choice allows us, in Section 4.7, to combine the algorithmic security result in this section with the implementation equivalence proofs [129] (again with minor adaptations) and connect the security theorems we prove here to concrete assembly implementations of ML-KEM.

4.6.1 Instantiating FO_K with K-PKE

We first instantiate the results on FO_K obtained in Section 4.5 with those obtained in Section 4.4. This instantiation is straightforward in EasyCrypt, which easily supports purely modular results. Note, however that, at this point, correctness is still conditional on the assumptions about the encoding and decoding operators introduced in Section 4.4.

Theorem 5. *Let δ be the maximum probability that experiment* CorrectnessBound *returns* 1*, and ϵ_{kg} and ϵ_{enc} denote the pseudorandomness advantage against the derandomizing of key generation and encryption algorithms in* K-PKE*, as defined*

in Section 4.4. Then, $\mathsf{FO}_K[\mathsf{K\text{-}PKE}, \mathsf{J}]$ *is* $(\delta + \epsilon_{\mathsf{kg}} + \epsilon_{\mathsf{enc}})$*-correct. Furthermore, let* $\epsilon_{\mathsf{MLWE_H}}$ *denote the maximum advantage of the reductions against Hashed MLWE constructed Section 4.4.*[26]

Then for any IND-CCA *adversary* $\mathcal{B}$ *against* $\mathsf{FO}_K[\mathsf{K\text{-}PKE}, \mathsf{J}]$*, issuing at most* q_D *queries to the decapsulation oracle, and at most* q_{GH} *queries to its random oracle* GH*, there exists a* PRF *adversary* $\mathcal{D}$ *against* J *such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{FO}_K[\mathsf{K\text{-}PKE},\mathsf{J}]}(\mathcal{B}) \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{J}}(\mathcal{D}) + Q \cdot (\delta + \epsilon_{\mathsf{kg}} + \epsilon_{\mathsf{enc}}) + \frac{2(2q_{\mathsf{GH}} + 2)}{|\mathcal{M}|} + 8 \cdot \epsilon_{\mathsf{MLWE_H}}$$

for $Q = 3 \cdot (2q_{\mathsf{GH}} + 3) + 5$ *and the running times of the reductions to Hashed MLWE and* $\mathcal{D}$ *are about that of* $\mathcal{B}$*.*

4.6.2 Connection to the ML-KEM specification

The ML-KEM-768 specification can be found in the artifact that accompanies this paper under `crypto-specs/mlkem`. It is, for the most part, identical to the specification of Kyber described in detail in [129]. The only difference resides in the file that implements the top-level KEM construction, where the hashing pattern of the Fusijaki-Okamoto transform was modified to match the change introduced in the ML-KEM draft standard and briefly discussed in Section 4.3.2. Let us denote the EasyCrypt module that fully specifies ML-KEM by MLKEM and note that the key generation and encryption algorithms are given in derandomized form, i.e., taking all randomness as explicit additional input. We obtain a security theorem that applies to this specification in four steps:

1. Instantiate Theorem 5 by making concrete all remaining abstractions, and discharging assumptions. These proofs reuse some lemmas on the NTT and other operations from [129].

2. Define an alternative version of MLKEM, which we call MLKEM_Op below, where we introduce two modifications: 1) the hash function used in the FO transform is turned into an oracle, aligning with the GH oracle in FO_K; and 2) the rejection sampling procedures for the matrix A used in key generation and encapsulation are replaced by a deterministic function H.[27]

3. Prove that, when H is implemented as in ML-KEM, and when GH is instantiated with SHA3-512, MLKEM_Op is the same as MLKEM. This step

[26] In this section we simplify the theorem statements by using quantities such as $\epsilon_{\mathsf{MLWE_H}}$ to denote the maximum advantage among a small number of fully defined reductions to the same assumption. They do not represent the computation of a maximum advantage over all adversaries in a class.

[27] Note that we see SHA3-512 as a random oracle only for the fixed input size used in the FO transform, which includes a message and the hash of the public-key. When SHA3-512 is used in K-PKE with a different input size—to smooth the key generation randomness—we model it as a stateless, fixed-length, pseudorandom generator.

```
op pk_encode(pk : W8.t Array32.t * polyvec) : pkey =
                                  (encode12_vec (toipolyvec (nttv pk[2])), pk[1]).
op pk_decode(pk : pkey) = (pk[2], invnttv (ofipolyvec (sem_decode12_vec (pk[1])))).
op sk_encode(sk : polyvec) : skey = encode12_vec_aux (toipolyvec (nttv sk)).
op sk_decode(sk : skey) = invnttv (ofipolyvec (sem_decode12_vec sk)).
op m_encode(m : plaintext) : poly = decompress_poly 1 (sem_decode1 m).
op m_decode(p : poly) : plaintext = encode1 (compress_poly 1 p).
op c_encode(c : polyvec * poly) : ciphertext =
    (encode10_vec (compress_polyvec 10 c[1]), encode4 (compress_poly 4 c[2])).
op c_decode(c : ciphertext) =
    (decompress_polyvec 10 (sem_decode10_vec c[1]),
                                                  decompress_poly 4 (sem_decode4 c[2])).
op rnd_err_v = compress_poly_err 4.
op rnd_err_u = mapv (compress_poly_err 10).
op max_noise = q %/ 4 − 1.
op under_noise_bound (p : poly) (b : int) = all (fun cc => | as_sint cc| ≤ b) p.
op cv_bound_max : int = 104.
```

Figure 4.11: Snippet of the instantiation of Theorem 5.

clarifies how our results apply to MLKEM: GH is modeled as a random oracle, and the matrix sampling procedure fixes the concrete parameters of the Hashed MLWE assumption.

4. Prove that MLKEM_Op is a secure KEM when the random coins of key generation and key encapsulation are sampled uniformly at random. This is done by showing that the advantage of any attacker in breaking MLKEM_Op in the ROM can be bounded using (the concrete version of) Theorem 5.

We now discuss each of these steps next.

4.6.2.1 Refining Theorem 5.

We refine Theorem 5 via theory cloning and parameter instantiation: some of the types and operators left abstract in Section 4.4 are given concrete definitions, and all axioms about them are discharged. The abstract parameters we instantiate here include the polynomial ring, matrix and vector dimensions, the types of keys, ciphertexts, shared keys, etc., the randomness distributions and the randomness expansion operators, the compression and serialization procedures, the noise thresholds required for bounding the failure probability, and the type of the RO.

Figure 4.11 shows (some) of the concrete instantiations. For example, the pk_encode and sk_encode operations make it explicit that public keys and secret keys are encoded in the NTT domain, so proving the correctness of deserialization implies reasoning about the cancellation of the NTT. Important

lemmas discharged at this point include the proof that rnd_err_v and rnd_err_u correctly describe compression errors as additive noise, the proof that decode_msg correctly recovers an encoded message as long as all the coefficients are affected by a noise value under $q/4$, and the proof that the maximum rounding error occurring in a coefficient of v is cv_bound_max = 104.

4.6.2.2 MLKEM_Op and Relation to MLKEM.

Figure 4.12 shows MLKEM_Op, an operator-based ML-KEM specification that we use to connect the security theorem we obtain above to the ML-KEM specification. We prove that MLKEM_Op is equivalent to MLKEM when H is defined as the semantics of the matrix sampling procedure specified by MLKEM, and when the random oracle passed to MLKEM_Op is instantiated with SHA3-512.

The proof is mostly boiler plate; it implies providing functional descriptions of serialization routines and randomness expansion procedures and then proving that these operators are correctly expressing the semantics of the corresponding MLKEM procedures. This proof applies to the derandomized entry points kg_derand, enc_derand, and dec. However, when we analyze correctness and security of MLKEM_Op, we use the entry points kg and enc for key generation and encapsulation that simply sample the necessary coins uniformly at random before calling the derandomized entry points.

4.6.2.3 Security Theorem for MLKEM_Op.

Having defined MLKEM_Op, we can prove the following theorem.

Theorem 6 (Main Theorem). *Define for notation conciseness the following quantities:*

- *let* δ *be the maximum probability that experiment* CorrectnessBound *returns* 1
- *let* ϵ_{prf} *denote the pseudorandomness advantage against the PRF used in the noise sampling procedure of ML-KEM, which is SHAKE-256 with a fixed output size of* 128 *bytes,*
- *let* ϵ_{J} *denote the pseudorandomness advantage against the PRF used in the implicit rejection step of ML-KEM, which is SHAKE-256 with a fixed output size of* 32 *bytes,*
- *let* ϵ_{hs} *denote the pseudorandomness advantage against the smoothing function* G_coins *used in ML-KEM key generation, and*
- *let* $\epsilon_{\mathsf{MLWE_H}}$ *denote the advantage against Hashed MLWE, when* H *is defined as the matrix rejection sampling procedure defined by ML-KEM.*

```
module InnerPKE_Op = {
  proc kg_derand(coins: W8.t Array32.t)
          : pkey * skey = {
    (rho,s,e) <- prg_kg coins;
    a <- nttm (H rho); s <- nttv s; e <- nttv e;
    t <- ntt_mmul a s + e;
    tv <- encode12_vec_aux (toipolyvec t);
    sv <- encode12_vec_aux (toipolyvec s);
    return ((tv,rho),sv); }

  proc enc_derand(pk : pkey, m : plaintext,
                  r : W8.t Array32.t) : ciphertext = {
    (rv,e1,e2) <- prg_enc r; (tv,rho) <- pk;
    thati <- decode12_vec_aux tv;
    that <- ofipolyvec thati;
    aT <- nttm (trmx (H rho));
    rhat <- nttv rv;
    u <- invnttv (ntt_mmul aT rhat) + e1;
    mp <- decode1 m;
    v <- invntt (ntt_dotp that rhat) +
           e2 + decompress_poly 1 mp;
    c1 <- encode10_vec_aux (compress_polyvec 10 u);
    c2 <- encode4 (compress_poly 4 v);
    return (c1,c2); }

  proc dec(sk : skey, cph : ciphertext)
                : plaintext option = {
    (c1,c2) <- cph;
    ui <- decode10_vec_aux c1;
    u <- decompress_polyvec 10 ui;
    vi <- decode4 c2;
    v <- decompress_poly 4 vi;
    si <- decode12_vec_aux sk;
    s <- ofipolyvec si;
    mp <- v + ((-) (invntt (ntt_dotp s (nttv u))));
    m <- encode1 (compress_poly 1 mp);
    return Some m; }
}.
```

```
module (MLKEM_Op : Scheme) (O : POracle) = {
  proc kg_derand(coins : W8.t Array32.t * W8.t Array32.t)
          : publickey * secretkey = {
    kgs <- coins[1]; z <- coins[2];
    (pk, sk) <@ InnerPKE_Op.kg_derand(kgs);
    hpk <- H_pk pk;
    return (pk, (sk, pk, hpk, z));
  }

  proc kg() : publickey * secretkey = {
    coins <$- srand; k <$- srand;
    (pk, sk) <@ kg_derand((coins,k));
    return (pk,sk); }

  proc enc_derand(pk : publickey, coins : W8.t Array32.t)
                  : ciphertext * sharedsecret = {
    m <- coins; hpk <- H_pk pk;
    (_K, r) <@ O.get(m,hpk);
    c <@ InnerPKE_Op.enc_derand(pk, m, r);
    return (c, _K); }

  proc enc(pk : publickey)
          : ciphertext * sharedsecret = {
    coins <$- srand; (c, _K) <@ enc_derand(pk,coins);
    return (c,_K); }

  proc dec(sk : secretkey, cph : ciphertext)
                : sharedsecret option = {
    (skp, pk, hpk, z) <- sk;
    m <@ InnerPKE_Op.dec(skp, cph);
    (_K, r) <@ O.get(oget m,hpk);
    _K' <- J z cph;
    c <@ InnerPKE_Op.enc_derand(pk, oget m, r);
    if (c <> cph) _K <- _K';
    return (Some _K);
  }
}.
```

Figure 4.12: Operator-based version of the ML-KEM specification.

Then, MLKEM_Op *is* $(\delta + \epsilon_{\mathsf{hs}} + 2\epsilon_{\mathsf{prf}})$*-correct. Furthermore, for any* IND-CCA *adversary* $\mathcal{B}$ *against* MLKEM_Op*, issuing at most* q_D *queries to the decapsulation oracle, and at most* q_{GH} *queries to its random oracle* GH*, we have that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{MLKEM_Op}}(\mathcal{B}) \le \epsilon_{\mathsf{J}} + Q \cdot (\delta + \epsilon_{\mathsf{hs}} + 2\epsilon_{\mathsf{prf}}) + \frac{2(2q_{\mathsf{GH}} + 2)}{|\mathcal{M}|} + 8 \cdot \epsilon_{\mathsf{MLWE_H}}$$

for $Q = 3 \cdot (2q_{\mathsf{GH}} + 3) + 5$*,* $|\mathcal{M}| = 2^{256}$ *and the running times of all reductions are about that of* $\mathcal{B}$*.*

Sketch. The proof of this theorem in EasyCrypt has two parts. The first part is technically a simple equivalence. We show that any attacker $\mathcal{B}$ against MLKEM_Op is actually executing in an environment which is identical to that of an attacker against (the instantiated version of) Theorem 5.

Again, proving this step is mostly boiler-plate, showing that the operations on both experiments exactly match semantically, even though they may be written in a slightly different way.

The second part is a bit more challenging and much more interesting. Note that Theorem 5 talks about the pseudorandomness properties of operators prg_kg and prg_enc that expand the coins passed to key generation and encapsulation to approximate the distributions specified by the underlying Hashed MLWE assumption. However, the theorem we are proving states the final advantage expression as a function of the pseudorandomness security bounds offered by SHAKE-256. The prg_kg and prg_enc operators are defined as follows:

```
op prg_kg(coins : W8.t Array32.t) : W8.t Array32.t * polyvec * polyvec =
      ((G_coins coins)[1],
      offunv (fun i => cbd2sample (PRF (G_coins coins)[2] (W8.of_int i))),
      offunv (fun i => cbd2sample (PRF (G_coins coins)[2] (W8.of_int (i + 3))))).
op prg_enc(coins : W8.t Array32.t) : polyvec * polyvec * poly =
      (offunv (fun i => cbd2sample (PRF coins (W8.of_int i))),
      offunv (fun i => cbd2sample (PRF coins (W8.of_int (i + 3)))),
      cbd2sample (PRF coins (W8.of_int 6))).
```

They are the functional counterparts of the procedures in Figure 4.13, which we use in an intermediate step of the proof that MLKEM_Op and MLKEM are equivalent (i.e., in the first part of this proof). We omit from Figure 4.13 the details of procedure CBD2_PRF(PRF).sample that specifies the procedure CBD_η procedure introduced in Section 4.3 for $\eta = 2$. However, it was proved in [129] that this procedure generates the correct distribution of binomial noise when the output of PRF is replaced with uniform random bytes on each call. This allows us to show that the sampling procedures in Figure 4.13, when fed with uniform byte strings, generate distributions that are computationally close to those specified by operators prg_kg_ideal and prg_enc_ideal in Section 4.4. The proof for prg_enc is a direct reduction to the pseudorandomness of PRF, whereas the proof for prg_og requires us to first replace the output of G_coins with random byte strings, which gives us a uniform rho, and then mimic the prg_enc proof to justify the noise distribution. To the best of our knowledge, these low-level details of the operation of ML-KEM, although intuitive, have never been laid down in a concrete security bound. In our theorem, these are accounted for as the $(\epsilon_{\mathsf{hs}} + 2\epsilon_{\mathsf{prf}})$ terms, which make it explicit that a randomness expansion advantage term must be accounted for every time K-PKE correctness and security are used in the FO proof. ■

4.7 Connecting to ML-KEM implementations

Provably secure specifications are good; provably secure implementations are better. Following [129] and the methodology first outlined in [19], we 1)

```
proc prg_kg(coins:W8.t Array32.t)
      : W8.t Array32.t * polyvec * polyvec = {
  (rho, noiseseed) <- G_coins coins;
  PRF.k <- noiseseed;
  _N <- 0;  i <- 0;
  while (i < kvec) {
    c <@ CBD2_PRF(PRF).sample(_N);
    noise1 <- noise1[i<-c];
    _N <- _N + 1; i <- i + 1; }
  i <- 0;
  while (i < kvec) {
    c <@ CBD2_PRF(PRF).sample(_N);
    noise2 <- noise2[i<-c];
    _N <- _N + 1; i <- i + 1;   }
  return (rho,noise1,noise2);
}
```

```
proc prg_enc(noiseseed:W8.t Array32.t)
            : polyvec * polyvec * poly = {
  PRF.k <- noiseseed;
  _N <- 0; i <- 0;
  while (i < kvec) {
    c <@ CBD2_PRF(PRF).sample(_N);
    noise1 <- noise1[i<-c];
    _N <- _N + 1;  i <- i + 1; }
  i <- 0;
  while (i < kvec) {
    c <@ CBD2_PRF(PRF).sample(_N);
   noise2 <- noise2[i<-c];
    _N <- _N + 1; i <- i + 1; }
  e2 <@ CBD2_PRF(PRF).sample(_N);
  return (noise1,noise2, e2);
}
```

Figure 4.13: Randomness expansion procedures for ML-KEM.

adapt the KYBER implementation from [129] to implement ML-KEM; 2) prove that the resulting implementation is functionally correct with respect to the ML-KEM specification from Section 4.6; and 3) prove that the implementation is "cryptographic constant-time". This last task is carried out using Jasmin's type system, which ensures that well-typed code does not branch on secrets, does not access memory at secret locations, and uses basic operations which are not constant time—like division and modulo—on public arguments only. We note that our security proof allows us to clarify the assumptions that justify constant-time security of the implementations.[28] We can therefore state that the implementations are *as secure* as the ML-KEM specification itself: an adversary interacting with the implementation can be converted into one that attacks the specification with the same success probability, as long as the attacker is not exploiting implementation leakage beyond what we consider in the constant-time paradigm.

Finally, we test that the test vectors produced by these implementations match those produced by the reference implementation of the draft standard available from the `pq-crystals/kyber` code package[29] and most of the more extensive test vectors that were made available by Schmieg[30]. The only mismatch between the test vectors by our implementations and the ones by Schmieg's implementation are relating to public-key validation, which Schmieg implements, but we deliberately do not (see Section 4.3.2). Also, the C function-call ABI

[28] For example, our proof of constant-time explicitly requires us to declassify the public seed ρ that gives rise to the matrix A. This σ is generated as $(\rho, \sigma) \leftarrow \mathrm{G}(d)$ in key generation. Here, σ is used to generate the secret key, so clearly d cannot be public. In fact, the only way to justify this declassify statement is by looking at the security proof and observing that G is modeled as a PRG, and so ρ can be considered to be independent of σ and d under that assumption.

[29] `https://github.com/pq-crystals/kyber/blob/standard/`

[30] See `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/aCAX-2QrUFw/m/hy5gwcESAAAJ`.

used by the Jasmin implementations passes keys trough pointers; it is the caller's responsibility to ensure that those pointers point to memory regions of appropriate length and the callee has no way of checking this. We thus skip the tests with keys and ciphertexts of wrong length from Schmieg's set of test vectors.

The Jasmin compiler is certified to preserve semantics through compilation. Hence, through our proof of equivalence between EasyCrypt specification and Jasmin implementations, we conclude that our specification of ML-KEM is correct with respect to the test vectors used by library developers. Dually, we conclude that the assembly implementations produced by the Jasmin compiler correctly implement the EasyCrypt specification of the ML-KEM standard, and thus satisfy IND-CCA security down to cryptographic assumptions.

5

SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange

This chapter is for all practical purposes identical to the paper *"SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange"* [102], co-authored with Phillip Gajland, Bor de Kock, Giulio Malavolta, and Peter Schwabe. Published at USENIX Security 2024.

Abstract. The advent of quantum computers has sparked significant interest in post-quantum cryptographic schemes, as a replacement for currently used cryptographic primitives. In this context, lattice-based cryptography has emerged as the leading paradigm to build post-quantum cryptography. However, all existing viable replacements of the classical Diffie-Hellman key exchange require additional rounds of interactions, thus failing to achieve all the benefits of this protocol. Although earlier work has shown that lattice-based Non-Interactive Key Exchange (NIKE) is theoretically possible, it has been considered too inefficient for real-life applications.

In this work, we challenge this folklore belief and provide the first evidence against it. We construct an efficient lattice-based NIKE whose security is based on the standard module learning with errors (M-LWE) problem in the quantum random oracle model. Our scheme is obtained in two steps: (i) A passively-secure construction that achieves a strong notion of correctness, coupled with (ii) a generic compiler that turns any such scheme into an actively-secure one. To substantiate our efficiency claim, we provide an optimised implementation of our passively-secure construction in Rust and Jasmin. Our implementation demonstrates the scheme's applicability to real-world scenarios, yielding public keys of approximately 220 KBs. Moreover, the computation of shared keys takes fewer than 12 million cycles on an Intel Skylake CPU, offering a post-quantum security level exceeding 120 bits.

5.1 Introduction

A key exchange is a fundamental cryptographic primitive that allows two users to agree on a common secret key over an insecure channel, such as the Internet. When the protocol involves a single, asynchronous message from each party, it is known as a *Non-Interactive Key Exchange* (NIKE). The seminal work of Diffie and Hellman [83] introduced the well-known NIKE scheme that marked the birth of public-key cryptography; each party sends a single group element g^x (or g^y, respectively) and the shared key can be derived by computing $(g^y)^x = (g^x)^y$. From a theoretical stand-point NIKE implies the existence of public key encryption (PKE), key encapsulation mechanism (KEM), and even authenticated key-exchange (AKE) when combining the results of [68] with [98]. Moreover, in practice, the Diffie-Hellman key exchange lies at the heart of widely-used protocols such as Transport Layer Security (TLS) [183], the Signal protocol [158, 159], or the Noise protocol framework [174].

The looming threat of quantum computers, combined with the discovery of efficient quantum algorithms for factoring integers and computing discrete logarithms [194], has necessitated the exploration of alternative solutions based on new mathematical structures, departing from protocols based on the Diffie-Hellman key exchange. In particular, lattice-based cryptography [182] has emerged as the leading paradigm for constructing *post-quantum* cryptographic schemes. Notably, three out of four algorithms selected for standardisation by NIST are lattice-based [191, 153, 179].

While efficient lattice-based key exchange protocols exist [66, 11, 191], they are all qualitatively different from the standard Diffie-Hellman-style key exchange, in the sense that they require *additional rounds of interaction*. For many applications, where interaction is already built-in, these protocols are perfectly fine substitutes for Diffie-Hellman (which is not post-quantum secure). However, in many scenarios of interest, the non-interactive nature of NIKE protocols is crucial (we discuss concrete examples in further detail in Section 5.1.1). Unfortunately, despite almost two decades of research on the subject, an efficient lattice-based NIKE remains elusive. Perhaps more worryingly, a recent work [112] has shown theoretical barriers on the efficiency of lattice-based NIKE, calling into question whether it is even possible to build an efficient scheme at all. Thus, the current state of affairs, leaves open the following question:

**Is lattice-based *non-interactive*
key exchange feasible in practice?**

In our work we seek to answer this question in the affirmative, and show that lattice-based NIKE can be made efficient enough to be used in practice, whilst maintaining post-quantum security.

5.1.1 NIKE vs. KEMs and Applications

While the Diffie-Hellman (DH) key exchange happens to be non-interactive, most post-quantum approaches to key exchange are interactive key-encapsulation mechanisms (KEMs). In a NIKE, any user A can derive a shared key k_{AB} using their secret key sk_A and the public key pk_B of user B. At the same time, and without interaction with A, user B can compute the same shared key k_{AB} by combining their secret key sk_B with the public key pk_A of user A. On the other hand, in a KEM, this key-derivation becomes a two-stage, inherently asymmetric and interactive process. First, A invokes an encapsulation routine that accepts pk_B as input and produces as output the shared key k_{AB} and a ciphertext ct, which they send to B. User B then invokes the decapsulation routine that takes as input ct and secret key sk_B to produce the same shared secret k_{AB}. At this point, it is worth noting that KEMs should satisfy the standard security notions of IND-CCA security. The security models considered for NIKE are discussed in Section 5.4. Notably, we examine both a passive security notion and an active variant. In some protocols that employ DH, the non-interactive nature is not actually utilised, making the migration to post-quantum KEMs straightforward. Probably the most prominent example is TLS, which uses the DH key exchange with ephemeral keys on both sides for forward secrecy and has been updated to offer post-quantum security by using KEMs in multiple papers [67, 172, 48] and real-world deployments [141, 142, 140, 204].

However, other protocols *do* make use of the non-interactive nature of DH and their migration to post-quantum primitives is thus much more involved. A common pattern in these protocols is the use of static DH keys for authentication. One example is OPTLS by Krawczyk and Wee [135], a proposal that eliminates the need for handshake signatures in TLS. The idea was picked up in the post-quantum setting in the KEMTLS proposal by Schwabe, Stebila, and Wiggers [189]. Like OPTLS, also KEMTLS eliminates the need for handshake signatures, but unlike OPTLS uses static KEM keys for authentication. This adaptation comes at the cost of additional communication round-trips until full server authentication is achieved, which can be problematic in scenarios where protocols like HTTPS allow the server to transmit early payload data before completing the handshake. Similar issues with delayed authentication when moving from DH to KEMs were identified in the migration of WireGuard to the post-quantum setting in [124] and in the the recently proposed post-quantum version of the Noise protocol framework [22].

While these examples manage to migrate from DH to KEMs, they come at the additional cost of further communication round trips without requiring signature-based authentication or more complex cryptographic primitives. However, if communicating parties cannot be assumed to be online at the same time, this approach is doomed to fail. A prominent example of precisely this asynchronous communication setting is the Signal secure-messaging protocol and specifically the X3DH protocol [159] that is invoked when a user A starts

their communication with a (possibly offline) user B. The X3DH protocol uses a combination of ephemeral, static, and semi-static DH keys to achieve forward secrecy, mutual authentication, and offline deniability without the need for direct interaction between A and B. Several attempts have been made to migrate X3DH to the post-quantum setting [71, 114, 195, 85, 70], but they all either assume the existence of a reasonably efficient post-quantum NIKE, or fail to achieve the same security and privacy as the pre-quantum version from a single simple asymmetric primitive.

5.1.2 Our Contributions

In this work, we demonstrate the real-world feasibility of lattice-based *non-interactive key exchange*. We propose a new scheme, that we call "Swoosh", based on the hardness of the M-LWE problem. We show a proof of its security, both in the passive and active setting, and provide parameter sets for the former with over 120-bits of security against quantum adversaries (using the best known attacks that incorporate recent advances in lattice cryptanalysis). Our contributions can be summarised as follows.

1. We propose a new construction of NIKE based on the hardness of the M-LWE problem. Our construction is based on the standard template [84, 147], but with a new tweak that allows us to prove a strong notion of correctness (which, in turn, is necessary to achieve active security) in the quantum random oracle model (QROM). Somewhat interestingly, our use of the random oracle appears to be different from the Fiat-Shamir [96] and the Fujisaki-Okamoto [100, 101] transformations, and may thus be of independent interest.

2. We propose a compiler that allows for the generic transformation of a passively secure NIKE into an actively secure scheme using non-interactive zero-knowledge (NIZK) proofs, that satisfy a strong soundness property (simulation-sound online-extractability). While this approach is folklore, we provide, to the best of our knowledge, the first explicit treatment of this technique in the literature. Furthermore, the exact notion of passive security needed for the proof to go through, turns out to be surprisingly subtle to identify.

3. We provide a highly optimised implementation [31] of Passive-Swoosh, written in Rust and Jasmin [15, 18]. With carefully selected parameters, our scheme achieves more that 120 bits of security against *quantum* adversaries. Notably, our benchmarks reveal smaller public keys compared to the smallest parameter set of Classic McEliece [7], an interactive KEM selected for round 4 of the NIST-PQC competition. Furthermore, we demonstrate that Passive-Swoosh outperforms CSIDH [76], the only currently known

[31] See `https://github.com/MQuaresma/pswoosh`.

(and realistic) post-quantum NIKE by orders of magnitude in terms of speed. Together with existing NIZK proof libraries [148, 149], our work establishes the first competitive construction of a lattice-based NIKE for practical use.

5.1.3 Related work

Post-quantum NIKE. While interactive KEMs appear to be much more efficient in a post-quantum world than NIKEs, there have been notable efforts towards constructing post-quantum NIKE schemes as well. Boneh and Zhandry [62] showed a construction using indistinguishability obfuscation (iO) to construct a multiparty NIKE from pseudorandom generators. However, the practicality of this approach is hindered by the performance limitations of iO, rendering it mainly of theoretical interest. Much more practical was supersingular-isogeny Diffie-Hellman (SIDH) [126, 80]. However, in 2016, this construction was shown to be susceptible to active attacks [103]. This could be solved by employing the Fujisaki-Okamoto transform [100] in the NIST PQC candidate SIKE [127], but this came at the expense of turning the NIKE into an interactive KEM. Another approach to restoring the active security of SIKE was presented in [28]. This approach preserved the non-interactive nature of SIDH, but required many parallel protocol executions and thus massively increased computation time and message sizes. In 2022, all of these approaches based on SIDH were made obsolete by the numerous attacks against SIDH [74, 155, 185].

In 2018, Castryck, Lange, Martindale, Panny, and Renes proposed CSIDH, a different approach for constructing an isogeny-based NIKE [76]. CSIDH is not affected by the attacks on SIDH, and is arguably the most plausible candidate for practical post-quantum NIKE thus far, although the post-quantum security of concrete parameters is subject of debate [64, 49, 173]. Multiple works have considered the efficient and secure implementation of CSIDH, currently the fastest approach is a variant called CTIDH [29]. We provide a performance comparison of our proposal to CTIDH in Section 5.6.2. Notably, the work of [60] introduced a compiler for achieving passive to active security. However, the compiler assumes a base scheme with perfect correctness and thus it does not apply to lattice-based NIKE.

Lattice-based NIKE. The idea of lattice-based NIKE using the approach employed in Passive-Swoosh is not new; in [147] Lyubashevsky calls it *"folklore (since at least 2010)"*. An attempt at selecting parameters was made in [134]. However, the proposed scheme did not formally consider passive security, nor active security. Moreover, the selected parameters resulted in a correctness error that would not even allow the transformation into an actively secure scheme through the use of NIZK proofs, a crucial aspect we achieve in Swoosh.

In fact, prior to our work, lattice-based NIKE was widely considered impractical and this was even substantiated by theoretical evidence. The

work of [112] discovered information-theoretic barriers in constructing lattice-based NIKE with non-interactive reconciliations. In particular, they showed that any natural candidate of lattice-based NIKE with polynomial modulus-to-noise ratio would necessarily incur an inverse-polynomial correctness error. However, we stress that our work does not contradict the theorem of [112]. As the authors of [112] observe, non-interactive reconciliation is possible, if we consider (M-)LWE instances with *super-polynomial* modulus-to-noise ratio. This is indeed the regime of parameters that we adopt in our work.

5.2 Technical Outline

We give a self-contained overview of our approach for constructing a fast lattice-based NIKE. The following is somewhat informal and glosses over many important details, as it is only intended for an intuitive understanding of our approach. The reader is referred to the respective technical sections for precise statements.

The Basic Blueprint. Before delving into the specifics of our approach, it is useful to recall the folklore construction of lattice-based key exchange between Alice and Bob. Let $\boldsymbol{A}$ be a random public $N \times N$ matrix over some ring $\mathcal{R}_q$ and χ a noise distribution. The protocol proceeds as follows; Alice samples $\boldsymbol{s}_1$ and $\boldsymbol{e}_1$ from χ^N, and computes her public key as $\boldsymbol{s}_1^\top \boldsymbol{A} + \boldsymbol{e}_1^\top$. Bob samples an independent $\boldsymbol{s}_2$ and $\boldsymbol{e}_2$ from χ^N, and computes his public key as $\boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{e}_2$. After asynchronously obtaining each other's public keys, Alice and Bob can compute an approximate shared key as

$$\boldsymbol{s}_1^\top \left(\boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{e}_2\right) \approx \left(\boldsymbol{s}_1^\top \boldsymbol{A} + \boldsymbol{e}_1^\top\right) \boldsymbol{s}_2.$$

A simple calculation shows that the shared keys computed by both parties are identical with the exception of the error terms $\boldsymbol{s}_1^\top \boldsymbol{e}_2$ and $\boldsymbol{e}_1^\top \boldsymbol{s}_2$ for Alice and Bob, respectively. To correct these errors, known schemes in the literature run interactive *reconciliation* protocols, which can be realised quite efficiently. However, if we insist on a NIKE protocol, no further interaction is allowed, and Alice and Bob must correct the errors locally. That is, we need to devise a *non-interactive* reconciliation function $\mathtt{Rec}$ such that

$$\mathtt{Rec}\left(\boldsymbol{s}_1^\top \left(\boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{e}_2\right)\right) = \mathtt{Rec}\left(\left(\boldsymbol{s}_1^\top \boldsymbol{A} + \boldsymbol{e}_1^\top\right) \boldsymbol{s}_2\right).$$

Note that, thus far, we have assumed that both Alice and Bob compute their keys according to the specification of the protocol, i.e., we implicitly only considered passive attacks. However, for the security of the final scheme, it will be necessary to handle parties that may behave arbitrarily. In what follows, we show how we tackle these two challenges separately, in a way that preserves the efficiency and security of the scheme.

Challenge I: Non-Interactive Reconciliation. A natural approach for correcting the errors introduced by the noise terms, is to derive the key by *rounding* the coefficients of the resulting ring element. In fact this is the approach that we adopt in this work, however there are still new ideas required to simultaneously achieve all of the following objectives: (i) security from the hardness of the standard module learning with errors (M-LWE) problem, (ii) reducing the correctness error to negligible, and (iii) maintaining the concrete efficiency of the construction. Here, we stress that a negligible correctness error is not just a matter of convenience, but that a non-negligible correctness error translates to an attack against the scheme: Loosely speaking, this is because the attacker can observe whenever the key agreement fails, therefore learning some information about the secret key of the honest party. Let us now focus on making the rounding approach work for non-interactive reconciliation. A simple calculation shows that the error terms cause a correctness error, only when the term $\boldsymbol{s}_1^\top \boldsymbol{A}\boldsymbol{s}_2$ falls into a *danger interval*

$$S^* = \left[\frac{q}{4} \pm \beta^2 dN\right] \cup \left[\frac{3q}{4} \pm \beta^2 dN\right],$$

where β is a bound on the norm of the noise distribution and d is the degree of $\mathcal{R}_q$. It is tempting to conclude that, if q is sufficiently large, then this event only happens with negligible probability. However, this analysis is imprecise as it does not take into account *adaptive* attacks, where the adversary chooses their secret key intentionally to make this event more likely. To prevent this, and obtain a provably secure scheme, we add a *random shift* $\boldsymbol{r}$ to the term $\boldsymbol{s}_1^\top \boldsymbol{A}\boldsymbol{s}_2$ to ensure that their sum $\boldsymbol{s}_1^\top \boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{r}$ is indeed uniformly distributed in $\mathcal{R}_q$. Note that such $\boldsymbol{r}$ does not need to be kept private, although it is important that it is sampled independently of the keys. Our idea is to sample $\boldsymbol{r}$ as the output of a hash function (modelled as a random oracle) on input the two public keys. This allows us to achieve two goals simultaneously:

- Both parties can recompute the shift $\boldsymbol{r}$ without the need of further interaction.
- We can show that $\boldsymbol{s}_1^\top \boldsymbol{A}\boldsymbol{s}_2 + \boldsymbol{r}$ is indeed uniformly sampled, even if the adversary has quantum access to the random oracle.

In summary, we are able to build a non-interactive reconciliation mechanism so that the scheme is provably secure (in the passive settings) against the standard M-LWE assumption, in the QROM. In fact, we are also able to show a strong notion of correctness, namely that the adversary cannot cause a reconciliation error, *even if it is allowed to choose both secret keys*. This strong notion of correctness will be useful when lifting the scheme to the active setting.

Challenge II: From Passive to Active Security. The above discussion concerns keys that are guaranteed to be well-formed (passive security). However,

in real-world scenarios we have to deal with attackers that can behave arbitrarily. In the stronger notion of *active security* [73, 98] the adversary is given access to various oracles that allow them to register honest keys, register corrupt keys (ones to which they do not know the corresponding secret key), or reveal the shared key between an honest key and a corrupted one. Ultimately the adversary wins if he can distinguish between a random key and a shared key, that was derived from two honestly generated key pairs.

In order to prove the active security of our scheme we present a *compiler* that generically lifts our scheme to the active setting using non-interactive zero-knowledge (NIZK) proofs. Here it is crucial that our scheme satisfies the aforementioned strong notion of correctness, since the only thing that the NIZK guarantees is that the keys are in the support of the honest distributions, but otherwise they may be chosen arbitrarily. For technical reasons, we require a NIZK that satisfies the strong property of simulation-sound online-extractability. We refer the reader to Section 5.5 for more details.

Putting Everything Together. Overall, we obtain a passively secure construction in the QROM assuming the hardness of the Module-LWE (M-LWE) problem (for the active settings, we additionally require a NIZK proof). Compared to Ring-LWE (R-LWE), M-LWE gives us greater flexibility over the choice of parameters, when implementing our scheme. However, this introduces an additional complication: Unlike the case for R-LWE, where single polynomials are considered and their multiplication is commutative, in the case of M-LWE we work with matrices where the matrix multiplication is not generally commutative. For the general case of two parties without predefined roles in a protocol, there is no way to know ahead of time whether to left multiply or right multiply. This means that each public key is effectively duplicated by adding a left multiplied key and right multiplied key. However, we argue that in many cases, when parties have predefined roles in a protocol, such as a server or client, this issue can be resolved (the server could "go right" and the client "left" or vice versa). We defer a more detailed discussion of this to Section 5.5.3.

Our parameters are selected as to provide more than 120 bits of post-quantum security, taking into account recent advances in lattice cryptanalysis. We work over the ring $\mathcal{R}_q \coloneqq \mathbb{Z}_q[X]/(X^d + 1)$ with $d = 256$. Along with our public matrix $\boldsymbol{A} \in \mathcal{R}_q^{N \times N}$, where $N = 32$, this gives us a lattice dimension of 8192. In order to reduce the correctness error to reasonable levels, q had to be sufficiently large. We choose $q = 2^{214} - 255$, a prime that is simultanously NTT-friendly and close to a power-of-two making for more efficient field arithmetic. Furthermore, we use ternary noise sampled from a centred binomial distribution, for the sake of efficiency.

Finally, we provide an open-source implementation of Passive-Swoosh in Rust and Jasmin, which employs numerous optimisations rendering competitive benchmarks. Due to the modular fashion of our implementation we note that

it can easily be tailored to use different parameters or be incorporated with suitable NIZKs. We defer a more detailed discussion to Section 5.6.

5.3 Preliminaries

In this section we introduce our notation and review some quantum preliminaries along with the relevant lattice-based hardness assumptions.

5.3.1 Notation

We define some standard notation used throughout the paper.

Sets, Vectors, Polynomials and Norms. For integers a, b, where $a < b$, $[a, b]$ denotes the set $\{a, a+1, \dots, b\}$. For any positive $\beta \in \mathbb{Z}$, we define the set $[\beta] \coloneqq \{-\beta, \dots, -1, 0, 1 \dots, \beta\}$, and let $x \xleftarrow{\$} \mathcal{S}$ denote the uniform sampling of x from the set $\mathcal{S}$. Let $\mathbb{Z}_q$ denote the ring of integers modulo a prime q. We define $\mathcal{R} \coloneqq \mathbb{Z}[X]/(X^d + 1)$ to be the ring of integer polynomials modulo $X^d + 1$, for d a power of 2, and $\mathcal{R}_q \coloneqq \mathbb{Z}_q[X]/(X^d + 1)$ the ring of integer polynomials modulo $X^d + 1$ where each coefficient is reduced modulo q. Bold upper-case letters $\boldsymbol{A}$ and bold lower case letters with arrows $\boldsymbol{\vec{a}}$ denote matrices and column vectors over $\mathcal{R}_q$, respectively; for row vectors we use the transpose $\boldsymbol{b}^\top$. For a polynomial $\boldsymbol{f} \in \mathcal{R}_q$, let $\mathbf{f} \in \mathbb{Z}_q^d$ denote the coefficient vector of $\boldsymbol{f}$, and $f_i \in \mathbb{Z}_q$ the i^{th} coefficient. However, we denote the constant coefficient by $\tilde{f} \coloneqq f_0 \in \mathbb{Z}_q$. For an element $f_i \in \mathbb{Z}_q$, we write $|f_i|$ to mean $|f_i \bmod q|$. Let the ℓ_∞ norm for $\boldsymbol{f} = \sum_{i=0}^{d-1} f_i X^i \in \mathcal{R}_q$ and $\boldsymbol{f} = (\boldsymbol{f}_1, \dots, \boldsymbol{f}_k) \in \mathcal{R}_q^k$ be defined as

$$\|\boldsymbol{f}\|_\infty \coloneqq \max_{0 \le i \le d-1} |f_i| \quad \text{and} \quad \|\boldsymbol{f}\|_\infty \coloneqq \max_{1 \le i \le k} \|\boldsymbol{f}_i\|_\infty,$$

respectively.

Probabilities, Algorithms and Games. The support of a discrete random variable X is defined as $\sup(X) \coloneqq \{x \in \mathbb{R} : \Pr[X = x] > 0\}$. Algorithms are denoted by upper-case letters in sans-serif font, such as A and B. Unless otherwise stated all algorithms are probabilistic and $(x_1, \dots) \xleftarrow{\$} \mathsf{A}(y_1, \dots)$ is used to denote that A returns $(x_1, \dots)$ when run on input $(y_1, \dots)$. When A has oracle access to B during its execution, this is denoted by A^{B}. For a probabilistic algorithm A, the notation $x \in \mathsf{A}(y)$ denotes that x is a possible output of A on input y. We use code-based security games [43], where $\Pr[\mathsf{G} \Rightarrow 1]$ denotes the probability that the final output of game G is 1. The notation $[\![B]\!]$, where B is a Boolean statement, refers to a bit that is 1 if the statement is true and 0 otherwise. The following lemma demonstrates the high probability of a randomly selected matrix being invertible. Due to space constraints, the proof is deferred to Section 5.3.1.

Lemma (Invertibility of Random Matrices). For $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^d + 1)$ with $d = 256$ and $q = 2^{214} - 255$, if $\boldsymbol{A}$ is a random matrix sampled from $\mathcal{R}_q^{N \times N}$, then the probability of $\boldsymbol{A}$ being invertible, denoted by $\Pr[\boldsymbol{A} \in \mathsf{GL}_N(\mathcal{R}_q)]$, satisfies

$$\Pr[\boldsymbol{A} \in \mathsf{GL}_N(\mathcal{R}_q) \mid \boldsymbol{A} \xleftarrow{\$} \mathcal{R}_q^{N \times N}] \geq \left(1 - \frac{128}{q^2}\right)^N.$$

Proof. Let $\mathcal{R}_q \coloneqq \mathbb{Z}_q[X]/(X^d + 1)$ and let $\eta(f(X))$ denote the set of irreducible proper divisors of the polynomial $f(X) = X^d + 1 \in \mathbb{Z}_q$. Then

$$\begin{aligned}
&\Pr[\boldsymbol{A} \in \mathsf{GL}_N(\mathcal{R}_q) \mid \boldsymbol{A} \xleftarrow{\$} \mathcal{R}_q^{N \times N}] \\
&\quad = \frac{\mathrm{ord}(\mathsf{GL}_N(\mathcal{R}_q))}{q^{N^2 \cdot \deg(f)}} \\
&\quad \geq \prod_{k=0}^{N-1} \left(1 - q^{(k-N)\cdot \deg(f)} \sum_{\alpha \in \eta(f)} q^{(N-k)\cdot(\deg(f) - \deg(\alpha))}\right) \\
&\quad = \prod_{k=0}^{N-1} \left(1 - q^{(k-N)\cdot d} 128 q^{(N-k)\cdot(d-2)}\right) \\
&\quad = \prod_{k=0}^{N-1} \left(1 - 128 q^{2(k-N)}\right) \\
&\quad = \prod_{k=0}^{N-1} \left(\frac{q^{2N} - 128 q^{2k}}{q^{2N}}\right) \\
&\quad = \left(\frac{1}{q^{2N^2}} \left(\prod_{k=0}^{N-1} \left(q^{2N} - 128 q^{2k}\right)\right)\right) \\
&\quad \geq \frac{1}{q^{2N^2}} \left(q^{2N} - 128 q^{2(N-1)}\right)^N \\
&\quad = \left(1 - \frac{128}{q^2}\right)^N,
\end{aligned}$$

where the first inequality follows from Lemma 1. ∎

5.3.2 Quantum Preliminaries

We review some quantum preliminaries as stated in [92]. Additional preliminaries are deferred to Section B.1.

Quantum Random Oracle Model. In the random oracle model [42], all parties have access to a uniformly sampled random function H. Since quantum adversaries can evaluate hash functions in superposition, we model quantum adversaries to have quantum access to random oracles [63]. Specifically, we assume that all algorithms have access to the unitary implementing the mapping: $|x\rangle |y\rangle \mapsto |x\rangle |y \oplus \mathsf{H}(x)\rangle$ where H is a uniformly sampled random function.

Query Depth and Query Parallelism. As in the work of [20] we consider the query depth D of an adversary making a total of Q_{H} random oracle queries. This is important in practice because for highly parallel adversaries we have $D \ll Q_{\mathsf{H}}$. By setting $D \coloneqq Q_{\mathsf{H}}$ we obtain the bounds for sequential adversaries. We will use the following technical lemma from [20].

Lemma 1 (Search in unstructured functions [20, Lem. 2]). *Let* H *be a random function drawn from a distribution such that* $\Pr[\mathsf{H}(x) = 1] \leq \lambda$ *for all* x*. Let* A *be an adversary with query depth* D*, making at most* Q_{H} *many queries to* H*. Then*

$$\Pr\left[\mathsf{H}(x) = 1 : b \xleftarrow{\$} \mathsf{A}^{\mathsf{H}}\right] \leq 4 \cdot (D+2) \cdot (Q_{\mathsf{H}}+1) \cdot \lambda.$$

5.3.3 Hardness Assumption

The security of our scheme relies on Module-Learning With Errors (M-LWE), a well-known computational lattice problem [182, 144].

Definition 1 (M-LWE$_{q,n,m,\chi}$). The decisional *Module-Learning With Errors* problem (in its Hermite normal form) with parameters $n, m > 0$ and an error distribution χ over $\mathcal{R}_q$ is defined via the game $\mathbf{M\text{-}LWE}^b_{q,n,m,\chi}$ depicted in Figure 5.1. Here, $\mathbf{M\text{-}LWE}^b_{q,n,m,\chi}$ is parameterised by a bit b. We define A's advantage in $\mathbf{M\text{-}LWE}^b_{q,n,m,\chi}$ as

$$\mathsf{Adv}^{\mathbf{M\text{-}LWE}}_{q,n,m,\chi}(\mathsf{A}) \coloneqq \left| \begin{array}{l} \Pr[\mathbf{M\text{-}LWE}^{0,\mathsf{A}}_{q,n,m,\chi} \Rightarrow 1] \\ -\Pr[\mathbf{M\text{-}LWE}^{1,\mathsf{A}}_{q,n,m,\chi} \Rightarrow 1] \end{array} \right|,$$

and say that $\mathbf{M\text{-}LWE}_{q,n,m,\chi}$ is ϵ-hard for all adversaries A satisfying $\mathsf{Adv}^{\mathbf{M\text{-}LWE}}_{q,n,m,\chi}(\mathsf{A}) \leq \epsilon$.

Game M-LWE$^b_{q,n,m,\chi}$	**Oracle** $\mathtt{RoR}(b)$ // Once
01 $b' \xleftarrow{\$} \mathsf{A}^{\mathtt{RoR}(b)}$	03 $\boldsymbol{A} \xleftarrow{\$} \mathcal{R}_q^{n \times m}$
02 **return** $[\![b = b']\!]$	04 **if** $b = 0$:
	05 $\quad \boldsymbol{s} \xleftarrow{\$} \chi^m$
	06 $\quad \boldsymbol{e} \xleftarrow{\$} \chi^n$
	07 $\quad$ **return** $(\boldsymbol{A}, \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e})$
	08 **elseif** $b = 1$:
	09 $\quad \boldsymbol{u} \xleftarrow{\$} \mathcal{R}_q^n$
	10 $\quad$ **return** $(\boldsymbol{A}, \boldsymbol{u})$

Figure 5.1: Game defining $\mathbf{M\text{-}LWE}^b_{q,n,m,\chi}$ with adversary A.

Theoretic treatments of LWE-based schemes typically consider the modulus to be polynomial in n and χ to be the discrete Gaussian on $D_{\mathbb{Z},\alpha\cdot q}$ over $\mathbb{Z}$ with mean 0 and standard deviation $\sigma = \alpha \cdot q/\sqrt{2\pi}$ for some $\alpha < 1$. For these choices the work of [182, 69] showed that if $\alpha q > 2\sqrt{n}$ then worst-case GapSVP-$\tilde{\mathcal{O}}(n/\alpha)$

reduces to average-case LWE. As such, many early implementations sampled from a discrete Gaussian distribution, which turns out to be either fairly inefficient [67] or vulnerable to timing attacks [72, 175, 95]. Furthermore, the performance of the best known attacks against LWE-based encryption schemes does not depend on the exact distribution of noise, but rather on the standard deviation (and potentially the entropy). This motivates the use of noise distributions that we can easily, efficiently, and securely sample from. One example is the centred binomial distribution used by CRYSTALS-Kyber [191] and in [11].

5.4 Definitions

In this section we present a formal definition of a non-interactive key exchange along with its security notions. A precise definition of non-interactive zero-knowledge proofs can be found in Section B.2.1.

5.4.1 Non-Interactive Key Exchange

Following the work of [73, 98], we formally define a non-interactive key exchange (NIKE). Through the use of IDs, the security model proposed in [73] abstracts away all considerations concerning certification and public key infrastructure.

Definition 2 (Non-Interactive Key Exchange). A non-interactive key exchange NIKE is defined as a tuple $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ of the following PPT algorithms. Furthermore, we define an identity space $\mathcal{IDS}$ and a shared key space $\mathcal{SKS}$.

$par \xleftarrow{\$} \mathsf{Stp}(1^\lambda)$**:** Given the security parameter 1^λ (encoded in unary), the probabilistic setup algorithm returns a set of system parameters par.

$(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\mathtt{ID})$**:** Given an identity $\mathtt{ID} \in \mathcal{IDS}$, the probabilistic key generation algorithm Gen returns a secret/public key pair (sk, pk).

$k \leftarrow \mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2)$**:** Given an identity $\mathtt{ID}_1 \in \mathcal{IDS}$ and its corresponding public key pk_1 along with another identity $\mathtt{ID}_2 \in \mathcal{IDS}$ and its corresponding secret key sk_2, the deterministic shared key establishment algorithm SdK returns a shared key $k \in \mathcal{SKS}$, or a failure symbol $\perp$. We assume that SdK always returns $\perp$ if $\mathtt{ID}_1 = \mathtt{ID}_2$.

Correctness. Informally, *honest correctness* states that shared keys derived by two honest parties should be the same with overwhelming probability. Although our subsequent definition of correctness implies honest correctness, we state both definitions here for completeness.

Definition 3 (Honest Correctness). A non-interactive key exchange $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ has *honest correctness error* δ (or is said to be δ-*correct*), if for all $par \in \mathsf{Stp}(1^\lambda)$ and $\mathtt{ID}_1, \mathtt{ID}_2 \in \mathcal{IDS}$ it holds that,

$$\Pr\left[\mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2) \neq \mathsf{SdK}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, sk_1) \;\middle|\; \begin{matrix}(sk_1, pk_1) \xleftarrow{\$} \mathsf{Gen}(\mathtt{ID}_1) \\ (sk_2, pk_2) \xleftarrow{\$} \mathsf{Gen}(\mathtt{ID}_2)\end{matrix}\right] \leq \delta(\lambda),$$

where the probability is taken over the random choices of Stp and Gen.

In this work we define a stronger notion, *semi-malicious correctness* that captures the property that two maliciously chosen key pairs (that are in the support of the key generation algorithm) will not cause the key exchange to fail. Since this property clearly implies honest correctness, throughout the rest of this work we only focus on semi-malicious correctness. We formalise *semi-malicious correctness* for NIKE relative to a random oracle H via the game $\mathbf{SM\text{-}COR}_{\mathsf{NIKE}}$ depicted in Figure 5.2 and define the advantage of an adversary A in $\mathbf{SM\text{-}COR}_{\mathsf{NIKE}}$ as

$$\mathsf{Adv}^{\mathbf{SM\text{-}COR}}_{\mathsf{NIKE},par}(\mathsf{A}) \coloneqq \Pr[\mathbf{SM\text{-}COR}^{\mathsf{A}}_{\mathsf{NIKE}} \Rightarrow 1].$$

Definition 4 (Semi-malicious Correctness). Let $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ be a non-interactive key exchange. In the quantum random oracle model, we say that NIKE is $\delta(Q_\mathsf{H})$-**SM-COR** if for all $\mathtt{ID}_1, \mathtt{ID}_2 \in \mathcal{IDS}$ and for all (possibly unbounded) adversaries A of depth at most D, making at most Q_H queries (possibly in superposition) to the random oracle H, we have $\mathsf{Adv}^{\mathbf{SM\text{-}COR}}_{\mathsf{NIKE},par}(\mathsf{A}) \leq \delta(Q_\mathsf{H}, D)$. [32]

Game $\mathbf{SM\text{-}COR}_{\mathsf{NIKE}}$

$par \leftarrow \mathsf{Stp}(1^\lambda)$

$\left.\begin{matrix}\mathsf{supp}(\mathsf{Gen}(\mathtt{ID}_1)) \ni (sk_1, pk_1) \\ \mathsf{supp}(\mathsf{Gen}(\mathtt{ID}_2)) \ni (sk_2, pk_2)\end{matrix}\right\} \xleftarrow{\$} \mathsf{A}^{|\mathsf{H}\rangle}(par)$

return $[\![\mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2) \neq \mathsf{SdK}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, sk_1)]\!]$

Figure 5.2: Correctness game $\mathbf{SM\text{-}COR}_{\mathsf{NIKE}}$ for a non-interactive key exchange NIKE defined relative to a random oracle H with adversary A.

Passive Security. Following the conventions of [98], we formalise the notion of key indistinguishability with *passive security*, or honest key registration (HKR), for a non-interactive key exchange NIKE, with respect to system

[32] Note that in the standard model our correctness definition can be considered a special case where the number of random oracle queries is zero and hence $\delta(Q_\mathsf{H}, D)$ is a constant.

parameters $par \in \mathsf{Stp}(1^\lambda)$ via the game $\mathbf{HKR\text{-}CKS\text{-}l}_{\mathsf{NIKE},par}$ depicted in Figure B.1. In $\mathbf{HKR\text{-}CKS\text{-}l}_{\mathsf{NIKE},par}$, the adversary A may make two queries to the `RegHonUsr` oracle, where A provides and identity and the public and secret keys are derived honestly. A may then make one query to the `TestQue` oracle, where A has to distinguish the shared key from a random key. We define the advantage of adversary A in $\mathbf{HKR\text{-}CKS\text{-}l}_{\mathsf{NIKE},par}$ as $\mathsf{Adv}^{\mathbf{HKR\text{-}CKS\text{-}l}}_{\mathsf{NIKE},par}(\mathsf{A}) \coloneqq \left|\Pr\left[\mathbf{HKR\text{-}CKS\text{-}l}^{\mathsf{A}}_{\mathsf{NIKE},par} \Rightarrow 1\right] - \frac{1}{2}\right|$.

Definition 5 (Passive Security). Let $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ be a non-interactive key exchange. We say that NIKE is $(\epsilon, Q_{\mathsf{H}})$-**HKR-CKS-l** relative to $par \in \mathsf{Stp}(1^\lambda)$ if for all PPT adversaries A, making at most Q_{H} queries (possibly in superposition) to the random oracle H, two queries to the `RegHonUsr` oracle and one query to the `TestQue` oracle, we have $\mathsf{Adv}^{\mathbf{HKR\text{-}CKS\text{-}l}}_{\mathsf{NIKE},par}(\mathsf{A}) \leq \epsilon(Q_{\mathsf{H}})$.

Active Security. We formalise the notion of key indistinguishability with *active security* for a non-interactive key exchange NIKE, with respect to system parameters $par \in \mathsf{Stp}(1^\lambda)$ via the game $\mathbf{CKS}_{\mathsf{NIKE},par}$ depicted in Figure B.2. Observe that this **CKS** (sometimes called DKR-CKS, short for "Dishonest Key Registration") notion was first defined in [73] and is polynomially equivalent to **CKS-l** (where the adversary is only allowed to make two `RegHonUsr` queries and one `TestQue` query) and *m-CKS-heavy* in the work of [98]. Unsurprisingly our definition of active security implies the former notion of passive security. The game starts by selecting a bit b uniformly at random after which the adversary A is given access to four oracles. A's queries may be made adaptively and are arbitrary in number. The `RegHonUsr` and `RegCorUsr` oracles let A register honest and corrupted user public keys, respectively. A may make multiple queries to `RegCorUsr`, in which case only the most recent $(corrupt, \mathtt{ID}, \perp, pk)$ entry is kept. The `RevCorQue` oracle provides A with a shared key between a pair of registered identities, subject only to the restriction that at least one of the two identities was registered as honest. Depending on the bit b, the `TestQue` oracle returns either a random key or a shared key between two identities registered as honest. Finally, the adversary outputs a guess bit b' and wins the game if and only if $b = b'$. We define the advantage of adversary A in $\mathbf{CKS}_{\mathsf{NIKE},par}$ as $\mathsf{Adv}^{\mathbf{CKS}}_{\mathsf{NIKE},par}(\mathsf{A}) \coloneqq \left|\Pr\left[\mathbf{CKS}^{\mathsf{A}}_{\mathsf{NIKE},par} \Rightarrow 1\right] - \frac{1}{2}\right|$.

Definition 6 (Active Security [73]). Let $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ be a non-interactive key exchange. We say that NIKE is $(\epsilon, Q_{\mathsf{H}}, Q_{\mathtt{RHU}}, Q_{\mathtt{RCU}}, Q_{\mathtt{RCQ}}, Q_{\mathtt{TQ}})$-**CKS** secure relative to $par \in \mathsf{Stp}(1^\lambda)$ if for all PPT adversaries A making at most; Q_{H} queries (possibly in superposition) to the random oracle H, $Q_{\mathtt{RHU}}$ queries to `RegHonUsr`, $Q_{\mathtt{RCU}}$ queries to `RegCorUsr`, $Q_{\mathtt{RCQ}}$ queries to `RevCorQue`, and $Q_{\mathtt{TQ}}$ queries to `TestQue`, we have $\mathsf{Adv}^{\mathbf{CKS}}_{\mathsf{NIKE},par}(\mathsf{A}) \leq \epsilon$.

Single- and Multi-User Security. The following Theorem from [98] shows that **CKS-l** and **CKS** are polynomially equivalent and will become useful for our proofs in Section 5.5.2. The other direction of the Theorem is trivial.

Theorem 1 (CKS-light ⇒ CKS [98, Thm. 1]). For any adversary A against NIKE in the *CKS* model, there exists an adversary B that breaks NIKE in the *CKS-light* model such that

$$\mathsf{Adv}^{\mathbf{CKS}}_{\mathsf{NIKE},par}(\mathsf{A}) \leq \frac{Q^2_{\mathtt{RHU}} \cdot Q_{\mathtt{TQ}}}{2} \cdot \mathsf{Adv}^{\mathbf{CKS\text{-}l}}_{\mathsf{NIKE},par}(\mathsf{B}),$$

Furthermore, the following observation will also be useful for our proofs in Section 5.5.2. Note that a variant of the $\mathbf{CKS}_{\mathsf{NIKE},par}$ game in which the adversary A is only given access to the `RegHonUsr` and `TestQue` oracles (as well as the random oracle H) can be thought of as a multi-user version of the $\mathbf{HKR\text{-}CKS\text{-}l}_{\mathsf{NIKE},par}$ game. Naturally we call this game $\mathbf{HKR\text{-}CKS}_{\mathsf{NIKE},par}$, as in the work of [98], and define the advantage of adversary A analogous to the previous definitions. As noted in [98], Theorem 1 carries over to the HKR setting.

5.5 Construction

We present our NIKE construction in two steps by introducing a scheme that only satisfies passive security followed by a generic transformation that turns it into a scheme with active security.

5.5.1 Passive Setting

In this section we present our construction of a non-interactive key exchange with semi malicious correctness that satisfies key indistinguishability for honestly registered public keys (passive security) in the random oracle model. The scheme is depicted in Figure 5.3. The Gen algorithm generates two independent left and right secret keys sk_L and sk_R along with their respective public keys pk_L and pk_R. Subsequently, the SdK algorithm computes the shared key using $\mathtt{ID}_1$'s pk_L and $\mathtt{ID}_2$'s sk_R when $\mathtt{ID}_1 \leq \mathtt{ID}_2$ and vice versa. By Section 5.3.1 a random matrix over $\mathcal{R}_q$ will be invertible with overwhelming probability.

Correctness. In order to achieve better bounds in our proof of security, we show that our scheme satisfies both honest correctness as well as the stronger notion of semi-malicious correctness of Definition 3 and Definition 4, respectively. Although Theorem 2 implies Section 5.5.1, we will use the latter and state its proof in Section 5.5.1 for sake of completeness.

Lemma (Honest Correctness). For all (possibly unbounded) adversaries A the non-interactive key exchange $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ construction depicted

$\underline{\mathsf{Stp}(1^\lambda)}$

$\mathcal{R}_q \coloneqq \mathbb{Z}_q[X]/(X^d+1)$
$\boldsymbol{A} \xleftarrow{\$} \mathsf{GL}_N(\mathcal{R}_q)$
$par \coloneqq (q, d, \mathcal{R}_q, N, \boldsymbol{A})$
return par

$\underline{\mathsf{Gen}(\mathtt{ID})}$

$\boldsymbol{s}_L, \boldsymbol{s}_R \leftarrow \mathtt{Cbd}(\cdot)$ // Samples $\boldsymbol{s} \in \mathcal{R}_q^N$ from χ^N
$\boldsymbol{e}_L, \boldsymbol{e}_R \leftarrow \mathtt{Cbd}(\cdot)$ // Samples $\boldsymbol{e} \in \mathcal{R}_q^N$ from χ^N
$sk_L \coloneqq \boldsymbol{s}_L^\top \in \mathcal{R}_q^{1\times N}$
$sk_R \coloneqq \boldsymbol{s}_R \in \mathcal{R}_q^N$
$pk_L \coloneqq \boldsymbol{s}_L^\top \boldsymbol{A} + \boldsymbol{e}_L^\top \in \mathcal{R}_q^{1\times N}$
$pk_R \coloneqq \boldsymbol{A}\boldsymbol{s}_R + \boldsymbol{e}_R \in \mathcal{R}_q^N$
return $\left(sk_{\mathtt{ID}} \coloneqq (sk_L, sk_R), pk_{\mathtt{ID}} \coloneqq (pk_L, pk_R)\right)$

$\underline{\mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2)}$

if $\mathtt{ID}_1 \leq \mathtt{ID}_2$:
  $\boldsymbol{r} \coloneqq \mathsf{H}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, pk_2) \in \mathcal{R}_q$
  parse $pk_1 \to (pk_L, \bot) =: \boldsymbol{u}_L^\top \in \mathcal{R}_q^{1\times N}$
  parse $sk_2 \to (\bot, sk_R) =: \boldsymbol{s}_R \in \mathcal{R}_q^N$
  $\boldsymbol{k}' \coloneqq \boldsymbol{u}_L^\top \boldsymbol{s}_R + \boldsymbol{r} \in \mathcal{R}_q$
else :
  $\boldsymbol{r} \coloneqq \mathsf{H}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, pk_1) \in \mathcal{R}_q$
  parse $pk_1 \to (\bot, pk_R) =: \boldsymbol{u}_R \in \mathcal{R}_q^N$
  parse $sk_2 \to (sk_L, \bot) =: \boldsymbol{s}_L^\top \in \mathcal{R}_q^{1\times N}$
  $\boldsymbol{k}' \coloneqq \boldsymbol{s}_L^\top \boldsymbol{u}_R + \boldsymbol{r} \in \mathcal{R}_q$
$k \coloneqq \mathtt{Rec}(\boldsymbol{k}') \in \{0,1\}^d$
return k

$\underline{\mathtt{Rec}(\boldsymbol{k})}$

for $i \in \{0, \ldots, d-1\}$:
  $\mathrm{k_i} \coloneqq \mathtt{Rnd}(k_i) \in \{0,1\}$
return $k \in \{0,1\}^d$

$\underline{\mathtt{Rnd}(k_i)}$

if $\frac{q}{4} \leq k_i \leq \frac{3q}{4}$:
  return 1
else :
  return 0

$\underline{\mathtt{Cbd}(\cdot)}$

for $i \in \{1, \ldots, N\}$:
  for $j \in \{0, \ldots, d-1\}$:
    $a, b \xleftarrow{\$} \{0,1\}$
    $f_j \coloneqq a - b$
  $\boldsymbol{f}_i \coloneqq \sum_{j=0}^{d-1} f_j X^j$
return $\boldsymbol{f} \coloneqq (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_N)$

Figure 5.3: Construction of passively secure non-interactive key exchange $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ with functions $\mathtt{Rec} : \mathcal{R}_q \to \{0,1\}^d$, $\mathtt{Rnd} : \mathbb{Z}_q \to \{0,1\}$ and $\mathtt{Cbd} : \varnothing \to \mathcal{R}_q^N$, and random oracle $\mathsf{H} : \mathcal{IDS} \times \left(\mathcal{R}_q^{1\times N} \times \mathcal{R}_q^N\right) \times \mathcal{IDS} \times \left(\mathcal{R}_q^{1\times N} \times \mathcal{R}_q^N\right) \to \mathcal{R}_q$. Here $\mathsf{GL}_N(\mathcal{R}_q)$ denotes the set of invertible $N \times N$ matrices over $\mathcal{R}_q$.

in Figure 5.3 has *honest correctness error*

$$\delta \leq \frac{4\beta^2 d^2 N}{q}$$

as per Definition 3.

Proof. The proof strategy is similar to the proof of Theorem 2, except that we can bound the probability of any coefficient of $\boldsymbol{k}^\star$ being in the interval

$$S^\star = \left[\frac{q}{4} \pm \beta^2 dN\right] \cup \left[\frac{3q}{4} \pm \beta^2 dN\right]$$

by $\frac{4\beta^2 d^2 N}{q}$, with a union bound over all coefficients. ∎

We show that the scheme satisfies semi-malicious correctness in the quantum random oracle model.

Theorem 2 (SM-COR of NIKE). For all (possibly unbounded) adversaries A of depth D making at most Q_{H} queries (possibly in superposition) to the random oracle H, the non-interactive key exchange $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$ construction depicted in Figure 5.3 has *semi-malicious correctness error*

$$\delta(Q_{\mathsf{H}}, D) \leq 16 \cdot (D+2) \cdot (Q_{\mathsf{H}}+1) \cdot \frac{\beta^2 d^2 N}{q}$$

as per Definition 4, where β is a bound on the maximum absolute value of the support of χ.

Proof. We are going to prove that the adversary cannot cause an error in the key derivation, i.e., a mismatch between the derived keys, even if he is allowed to choose both secret keys from the support of the key generation algorithm. This trivially implies semi-malicious correctness. Let (sk_1, pk_1) and (sk_2, pk_2) be the pairs returned by the adversary. Without loss of generality we can consider $sk_1 = sk_L$ and $pk_2 = pk_R$, i.e. only "one side" of the key. A key mismatch occurs whenever

$$\begin{aligned}
\mathtt{Rec}\left(pk_L^\top sk_R + \boldsymbol{r}\right) &\neq \mathtt{Rec}\left(sk_L^\top pk_R + \boldsymbol{r}\right) \\
\mathtt{Rec}\left(\left(\boldsymbol{s}_L^\top \boldsymbol{A} + \boldsymbol{e}_L^\top\right)\boldsymbol{s}_R + \boldsymbol{r}\right) &\neq \mathtt{Rec}\left(\boldsymbol{s}_L^\top\left(\boldsymbol{A}\boldsymbol{s}_R + \boldsymbol{e}_R\right) + \boldsymbol{r}\right) \\
\mathtt{Rec}\Big(\underbrace{\boldsymbol{s}_L^\top \boldsymbol{A}\boldsymbol{s}_R + \boldsymbol{r}}_{\boldsymbol{k}^\star \in \mathcal{R}_q} + \boldsymbol{e}_L^\top \boldsymbol{s}_R\Big) &\neq \mathtt{Rec}\Big(\underbrace{\boldsymbol{s}_L^\top \boldsymbol{A}\boldsymbol{s}_R + \boldsymbol{r}}_{\boldsymbol{k}^\star \in \mathcal{R}_q} + \boldsymbol{s}_L^\top \boldsymbol{e}_R\Big),
\end{aligned}$$

where $\boldsymbol{r}$ is the output of the random oracle on both public keys and $\boldsymbol{e}_L$ and $\boldsymbol{e}_R$ are sampled from the noise distribution χ^N. By definition of the $\mathtt{Rec}$ function, this means that the term $\boldsymbol{e}_L^\top \boldsymbol{s}_R$ (or, equivalently, the term $\boldsymbol{s}_L^\top \boldsymbol{e}_R$) is causing a rounding error on one out of the d coefficients of $\boldsymbol{k}^\star$. We now bound the size of the largest coefficient of $\boldsymbol{e}_L^\top \boldsymbol{s}_R$ as

$$\left\|\boldsymbol{e}_L^\top \boldsymbol{s}_R\right\|_\infty = \left\|\sum_{i=1}^{N} \boldsymbol{e}_{L,i}\boldsymbol{s}_{R,i}\right\|_\infty \leq \sum_{i=1}^{N} \left\|\boldsymbol{e}_{L,i}\boldsymbol{s}_{R,i}\right\|_\infty \leq \beta^2 dN,$$

where the first inequality follows from the triangle inequality. The norm of $\boldsymbol{s}_L^\top \boldsymbol{e}_R$ can be bounded similarly. It follows that, in order for a key derivation error to occur, then at least one coefficient of $\boldsymbol{k}^\star$ must be in the following interval

$$S^\star = \left[\frac{q}{4} \pm \beta^2 dN\right] \cup \left[\frac{3q}{4} \pm \beta^2 dN\right].$$

If $\boldsymbol{k}^\star$ followed a uniform distribution, this would then happen with probability at most $\frac{d \cdot |S^\star|}{|\mathbb{Z}_q|} = \frac{4\beta^2 dN}{q}$. Next we define a function F that, on input two public

keys and two identities samples a uniform $\boldsymbol{r}$, it returns 1 if a key mismatch occurs, i.e.,

$$\mathtt{Rec}\left(pk_L^\top sk_R + \boldsymbol{r}\right) \neq \mathtt{Rec}\left(sk_L^\top pk_R + \boldsymbol{r}\right)$$

and 0 otherwise. The function checks this by (inefficiently) recovering the secret keys and comparing the results of the $\mathtt{Rec}$ functions (see equation above). Due to the high probability of invertibility for $\boldsymbol{A}$, as stated in Section 5.3.1, the secret key is uniquely determined by the public key. Hence, this (inefficient) function is well defined on all inputs. Furthermore, note that the element

$$\boldsymbol{k}^\star = sk_L^\top \boldsymbol{A} sk_R + \boldsymbol{r}$$

is uniformly distributed in $\mathcal{R}_q$, since $\boldsymbol{r} \xleftarrow{\$} \mathcal{R}_q$. It follows that for any given input x:

$$\Pr[F(x) = 1] \leq \frac{4\beta^2 d^2 N}{q}.$$

Finally, observe that by definition a key mismatch happens if and only if the function F output 1 and consequently the adversary is able to find such accepting input. By Lemma 1, this happens with probability at most $16 \cdot (D+2) \cdot (Q_{\mathsf{H}} + 1) \cdot \beta^2 d^2 N/q$ for an adversary of depth D, making at most Q_{H} quantum queries to the random oracle. ■

On the Need for Random Oracles. An astute reader may wonder whether the usage of the random oracle is needed at all to prove the above notion of correctness, since there does not appear to be an immediate attack even if we omit the random oracle completely from the scheme. It is plausible to conjecture that semi-malicious correctness holds even without the random oracle. Informally, semi-malicious correctness boils down to showing that, for a given public key $pk \in \mathcal{R}_q^N$, it is hard to find an $\boldsymbol{s} \in \mathcal{R}_q^N$ such that no coefficient of the product $\boldsymbol{s}^\top pk$ lies in the interval $S^\star$. Thus, the bound in these settings would require one to estimate the hardness of this version of the (inhomogenous) 1-dimensional short integer solution (SIS) problem. By relying on the random oracle heuristic, we are able to bypass this problem and obtain a construction in the QROM that is: (i) unconditionally correct in *any ring* and (ii) whose security is based on the well-established M-LWE problem. We leave the precise study of the hardness of this 1-dimensional variant of the SIS problem as ground for future work.

Passive Security. Assuming the hardness of M-LWE, Definition 1, we show that the scheme satisfies *passive security*, Definition 5, in the QROM.

Theorem 3 (Passive Security). For any PPT adversary A against $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$, depicted in Figure 5.5, making an arbitrary number of queries (possibly in superposition) to H, there exists PPT adversaries $\mathsf{B}_1, \mathsf{B}_2$ such that

$$\mathsf{Adv}^{\textbf{HKR-CKS-l}}_{\mathsf{NIKE},par}(\mathsf{A}) \leq \mathsf{Adv}^{\textbf{M-LWE}}_{q,N,N,\chi}(\mathsf{B}_1) + \mathsf{Adv}^{\textbf{M-LWE}}_{q,N,N+1,\chi}(\mathsf{B}_2) + \frac{4\beta d}{q}.$$

Proof of Theorem 3. Let A be an adversary against NIKE in the **HKR-CKS-l** game. Consider the sequence of games in Figure 5.4 that takes into account both independent left and right halves of the keys.

Games G_0, G_1, G_2, G_3

01 $b \xleftarrow{\$} \{0,1\}$
02 $(sk_1, pk_1) \xleftarrow{\$} \mathsf{Gen}(\mathtt{ID}_1)$
03 $pk_1 = (pk_{1_L}, pk_{1_R}) \xleftarrow{\$} \mathcal{R}_q^{1\times N} \times pk_{1_R}$ // G_1
04 $(sk_2, pk_2) \xleftarrow{\$} \mathsf{Gen}(\mathtt{ID}_2)$
05 $pk_2 = (pk_{2_L}, pk_{2_R}) \xleftarrow{\$} pk_{2_L} \times \mathcal{R}_q^N$ // G_3
06 **if** $\mathtt{ID}_1 \le \mathtt{ID}_2$:
07 $\quad \boldsymbol{r} \coloneqq \mathsf{H}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, pk_2) \in \mathcal{R}_q$
08 $\quad$ **parse** $pk_1 \to (pk_L, \perp) =: \boldsymbol{u}_L^\top \in \mathcal{R}_q^{1\times N}$
09 $\quad$ **parse** $sk_2 \to (\perp, sk_R) =: \boldsymbol{s}_R \in \mathcal{R}_q^N$
10 $\quad \boldsymbol{k'} \coloneqq \boldsymbol{u}_L^\top \boldsymbol{s}_R + \boldsymbol{r} \in \mathcal{R}_q$
11 **else** :
12 $\quad \boldsymbol{r} \coloneqq \mathsf{H}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, pk_1) \in \mathcal{R}_q$
13 $\quad$ **parse** $pk_1 \to (\perp, pk_R) =: \boldsymbol{u}_R \in \mathcal{R}_q^N$
14 $\quad$ **parse** $sk_2 \to (sk_L, \perp) =: \boldsymbol{s}_R^\top \in \mathcal{R}_q^{1\times N}$
15 $\quad \boldsymbol{k'} \coloneqq \boldsymbol{s}_R^\top \boldsymbol{u}_R + \boldsymbol{r} \in \mathcal{R}_q$
16 $k_0 \coloneqq \mathsf{Rec}(\boldsymbol{k'}) \in \{0,1\}^d$ // G_0
17 $\boldsymbol{e} \xleftarrow{\$} \chi$ // G_2
18 $k_0 \coloneqq \mathsf{Rec}(\boldsymbol{k'} + \boldsymbol{e}) \in \{0,1\}^d$ // G_2
19 $\boldsymbol{u} \xleftarrow{\$} \mathcal{R}_q$ // G_3
20 $k_0 \coloneqq \mathsf{Rec}(\boldsymbol{u}) \in \{0,1\}^d$ // G_3
21 $k_1 \xleftarrow{\$} \mathcal{SKS}$
22 $b' \leftarrow \mathsf{A}^{|\mathsf{H}\rangle}(pk_1, pk_2, k_b)$
23 **return** $[\![b = b']\!]$

Figure 5.4: Games G_0, G_1, G_2, G_3 for the proof of **HKR-CKS-l** of NIKE in Figure 5.3.

Game G_0 This is the original $\textbf{HKR-CKS-l}_{\mathsf{NIKE},par}$ game so by definition $\Pr\left[\mathsf{G}_0^{\mathsf{A}} \Rightarrow 1\right] = \Pr\left[\textbf{HKR-CKS-l}_{\mathsf{NIKE},par}^{\mathsf{A}} \Rightarrow 1\right]$.

Game G_1 Without loss of generality we consider the half of pk_1 that is actually used in the key derivation, say pk_{1_L}. In this game pk_{1_L} is replaced with a uniform key on Line 03. It follows immediately from Definition 1 that

$$\left|\Pr\left[\mathsf{G}_0^{\mathsf{A}} \Rightarrow 1\right] - \Pr\left[\mathsf{G}_1^{\mathsf{A}} \Rightarrow 1\right]\right| \le \mathsf{Adv}^{\textbf{M-LWE}}_{q,N,N,\chi}(\mathsf{B}_1).$$

Game G_2 In this hybrid we modify the way we compute the shared key. Consider $\boldsymbol{k'}$ as computed in the SdK algorithm, we define the shared key as $\mathsf{Rec}(\boldsymbol{k'} + \boldsymbol{e})$ where $\boldsymbol{e} \xleftarrow{\$} \chi$ is a freshly sampled ring element from the noise distribution. Note that the adversary can only detect a change in this hybrid if

$$\mathsf{Rec}(\boldsymbol{k'} + \boldsymbol{e}) \neq \mathsf{Rec}(\boldsymbol{k'}).$$

Since $\boldsymbol{k}'$ is uniformly distributed in $\mathcal{R}_q$, the probability that any coefficient is rounded to a different term is at most $4\beta d/q$, which is also an upper bound on the distinguishing advantage of the adversary. Thus we get

$$\left|\Pr\left[\mathsf{G}_1^{\mathsf{A}} \Rightarrow 1\right] - \Pr\left[\mathsf{G}_2^{\mathsf{A}} \Rightarrow 1\right]\right| \leq \frac{4\beta d}{q}.$$

Game G_3 In this game the half of pk_2 that is used in the key derivation, pk_{2_R}, is replaced with a uniform key on Line 05. Furthermore $\boldsymbol{k}' + \boldsymbol{e}$ is replaced with a uniform ring element $\boldsymbol{u}$ on Line 20. By an invocation of the module-LWE assumption we have that

$$\left|\Pr\left[\mathsf{G}_2^{\mathsf{A}} \Rightarrow 1\right] - \Pr\left[\mathsf{G}_3^{\mathsf{A}} \Rightarrow 1\right]\right| \leq \mathsf{Adv}^{\textbf{M-LWE}}_{q,N,N+1,\chi}(\mathsf{B}_2).$$

Observe that k_0 and k_1 are identically distributed and the adversary can only guess b'. Hence, $\Pr\left[\mathsf{G}_3^{\mathsf{A}} \Rightarrow 1\right] = \frac{1}{2}$. Collecting all probabilities yields the bound stated in Theorem 3.

■

Reconciliation Function. The reconciliation function $\mathtt{Rec}$ depicted in Figure 5.3 maps a ring element $\boldsymbol{k} \in \mathcal{R}_q$ to a bit string of length d. In the final step of the proof of Theorem 3 the input to $\mathtt{Rec}$ is a uniformly sampled ring element $\boldsymbol{u} \in \mathcal{R}_q$. Each element in $\mathbb{Z}_q$ is rounded to a bit with almost equal probability. More specifically for all $u_i \in \mathbb{Z}_q$, $\Pr[\mathtt{Rnd}(u_i) = 0] = \frac{q-1}{2q}$ and $\Pr[\mathtt{Rnd}(u_i) = 1] = 1 - \frac{q-1}{2q}$, which are negligibly close to 1/2, for an appropriate choice of q. Although the reconciliation function maps the key to a smaller domain, trying to guess $k \coloneqq \mathtt{Rec}(\boldsymbol{u})$, even when given both public keys, would, in expectation, require guessing half of all possible keys, which amounts to 2^{d-1}. This becomes practically infeasible for large values of d. For more details on our specific choice of q and d, we refer to Section 5.5.4.

5.5.2 Active Setting

Here we show how a non-interactive key exchange with passive security can be generically transformed to one with active security. The transformation, depicted in Figure 5.5, requires a simulation-sound NIZK with a straight line extractor. Due to space constraints, the proof is deferred to Section B.3.1.

Theorem 4 (HKR-CKS-l and SM-COR of NIKE' $\overset{\textbf{QROM}}{\underset{\mathsf{ZKPoK}}{\Rightarrow}}$ CKS of NIKE). Let $\mathsf{H} : \{0,1\}^* \to \mathcal{R}_q$ be a random oracle and $\mathsf{NIKE}' \coloneqq (\mathsf{Stp}', \mathsf{Gen}', \mathsf{SdK}')$ a passively secure non-interactive key exchange with semi-malicious correctness defined relative to $par' \in \mathsf{Stp}'(1^\lambda)$. Further, let $\mathsf{ZKPoK} \coloneqq (\mathsf{ZK.Prv}, \mathsf{ZK.Ver})$ be a simulation-sound online extractable zero-knowledge proof of knowledge for the NP relation $R = (pk_{\mathtt{ID}}, sk_{\mathtt{ID}})$. Then, for any **CKS** adversary A against

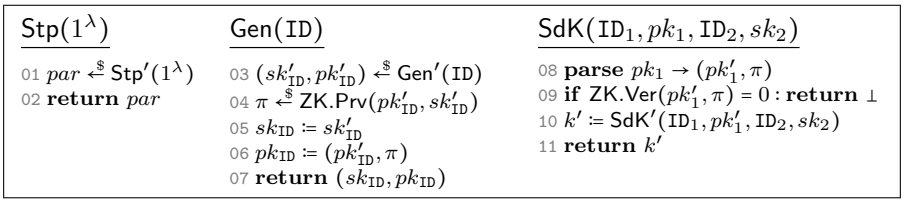


Figure 5.5: Compiler for transforming a passively secure non-interactive key exchange $\mathsf{NIKE}' \coloneqq (\mathsf{Stp}', \mathsf{Gen}', \mathsf{SdK}')$ with semi-malicious correctness into an actively secure non-interactive key exchange $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$.

$\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$, depicted in Figure 5.5, there exist PPT adversaries $\mathsf{B}_1, \mathsf{B}_2, \mathsf{B}_3, \mathsf{C}$ such that

$$\mathsf{Adv}^{\mathbf{CKS}}_{\mathsf{NIKE},par}(\mathsf{A}) \leq \frac{Q^2_{\mathtt{RHU}} \cdot Q_{\mathtt{TQ}}}{2} \cdot \epsilon$$

and

$$\begin{aligned}\epsilon \coloneqq\ & Q_{\mathtt{RCU}} \cdot \mathsf{Adv}^{\mathbf{SSND}}_{\mathsf{ZKPoK}}(\mathsf{B}_1) + 2 \cdot \mathsf{Adv}^{\mathbf{SM\text{-}COR}}_{\mathsf{NIKE}',par'}(\mathsf{B}_2) \\ & + 2 \cdot \mathsf{Adv}^{\mathbf{ZK}}_{\mathsf{ZKPoK}}(\mathsf{B}_3) + \mathsf{Adv}^{\mathbf{HKR\text{-}CKS\text{-}l}}_{\mathsf{NIKE}',par'}(\mathsf{C}),\end{aligned}$$

where $Q_{\mathtt{RHU}}$, $Q_{\mathtt{RCU}}$, $Q_{\mathtt{RCQ}}$, and $Q_{\mathtt{TQ}}$ denote the number of queries made by A to `RegHonUsr`, `RegCorUsr`, `RevCorQue`, and `TestQue`, respectively.

5.5.3 Practical considerations

Halving the Key Size. Observe that the "left" and "right" components pk_L and pk_R of the public key of the NIKE as specified in Figure 5.3 are necessary because we work in the non-commutative M-LWE setting. Note that all theorems, lemmas and proofs consider both the left and right components of keys. An easy way to halve the size of the public key would be to set $N = 1$, i.e., to work in the R-LWE setting; this also eliminates the need for the case distinction in SdK. We argue that for essentially all relevant applications of a NIKE, we can halve the public-key size even *without* moving to the R-LWE setting. All that is required is that protocol participants (and their associated NIKE keys) have different *roles*, typically called initiator and responder or client and server, and that these roles are clear from protocol context. This is certainly the case for the application examples sketched in Section 5.1.1: The OPTLS handshake, like the TLS handshake, clearly distinguishes the roles of client and server, so does the handshake in (post-quantum) WireGuard. Also in X3DH the critical static-semistatic key exchange has clear roles that can be used to distinguish between the "left" and "right" participant instead of transmitting both components of the key and using comparison of IDs. Note that this setting of a NIKE using keys with different roles is very similar to the ℓ_A and ℓ_B keys of

SIDH [126, Sec. 3.2], when it was still considered as a replacement for DH, i.e., before it was shown to not be actively secure in [103] and completely broken in [75].

Based on these considerations, we stick to the M-LWE setting for the construction of Swoosh; in our performance evaluation in Section 5.6 we report the size of only one public-key component.

Security of the NIZK. We highlight that our proof of active security, Theorem 4, requires the strong property of simulation-sound online-extractability. Although constructions satisfying such a strong notion exist [200], they tend to be less efficient than alternatives satisfying weaker notions of security. For instance, a proof of knowledge of an M-LWE secret satisfying simulation soundness, but without *online*-extractability, using state-of-the-art techniques [148] and appropriate parameters is around 89 KB in size. Since the [148] framework is relatively new, there are currently no implementations available to determine the exact running time of the prover and verifier. However, previous versions of lattice-based zero-knowledge proofs [149], which serve as the basis for [148], have shown implementations with prover and verifier running times on the order of milliseconds when proving similar relations. We remark that using NIZKs with slightly weaker security, in favour of a more efficient scheme, is a well-established heuristic and was already used in many works prior to ours, including [77, 86, 87, 104, 145, 156, 198]. However, this limitation is inherent in our approach, and investigating security under an alternative notion remains an interesting (and challenging) question. Tangentially, we also mention that for some applications, the performance of the NIZK does not affect the efficiency of the shared-key computation, since it can be verified once and for all for a given public key: In any scenario where the public keys are distributed by a PKI, the NIZK proof can be simply verified by the PKI upon the registration of the key, and then immediately discarded. The users would then trust the PKI to have verified the NIZK on their behalf. Note that this does not introduce any extra trust assumption, since the PKI is already trusted to provide the correct public key. In these scenarios, the efficiency of the NIZK only marginally impacts the overall system performance, and thus justifies ignoring the costs of the NIZK for shared-key computation.

5.5.4 Parameter selection

Selecting parameters for the scheme influences several aspects, most notably the correctness error and the hardness of M-LWE. In order to evaluate the security of our scheme we use the *Lattice-Estimator* tool [6, 177, 8], to estimate the memory and CPU operations required to perform various lattice attacks, including dual attacks, uSVP, the Coded-BKW attack, and solving using Gröbner bases with the Arora-GB attack. The estimator has been used to estimate the concrete security for all LWE and NTRU based candidates

of the NIST competition [9], and is regularly updated to include the latest developments in lattice cryptanalysis [33]. However, we also take into account practical considerations for the implementation when selecting our parameters, such as the use of ternary secrets and noise sampled from a centred binomial distribution. For our scheme with parameters $n = 8192, q = 2^{214} - 255$ and $\mathcal{X}$ a ternary distribution, we estimate the hardness of the M-LWE problem underlying Swoosh at 120 bits [34].

The other way to attack Swoosh is, for an active attacker, to try to produce failures. We consider a quantum attacker with a bounded query depth of $D = 2^{64}$ (i.e., what NIST considers to be "the approximate number of gates that current classical computing architectures can perform serially in a decade" [169, Sec. 4.A]) and a bound on the number of queries of 2^{120} (i.e., matching the hardness of the underlying lattice problem). Applying Theorem 2 yields a success probability (correctness error), after this amount of computation,

$$16 \cdot \left(2^{64} + 2\right) \cdot \left(2^{120} + 1\right) \cdot \frac{256^2 \cdot 32}{2^{214}} < \frac{1}{2^4} = \delta(Q_{\mathsf{H}}),$$

i.e., considerably smaller than $1/2$. Note that this analysis is conservative as it ignores the circuit depth for the Grover oracle that an attacker would need to implement.

Parameter	Description	Value
β	upper bound on $\|\boldsymbol{s}\|_\infty = \|\boldsymbol{e}\|_\infty$	1
q	prime modulus	$2^{214} - 255$
d	dim of $\mathcal{R}_q \coloneqq \mathbb{Z}_q[X]/(X^d + 1)$	256
l	# factors $X^d + 1$ splits into mod q	128
N	height of the $\boldsymbol{A}$ matrix	32
n	lattice dimension	8192
χ	secret / noise distribution	$p(-1) = 25\%$ $p(0) = 50\%$ $p(1) = 25\%$

Table 5.1: Parameter selection for non-interactive key exchange NIKE.

5.6 Implementation & Performance Evaluation

In order to demonstrate the efficiency of Swoosh in terms of performance, we implement the core part of the scheme, Passive-Swoosh, present benchmarks of this implementation, and compare to other KEMs and (pre- and post-quantum)

[33] An up-to-date list of implemented works can be found `https://lattice-estimator.readthedocs.io/en/latest/references.html`.

[34] These numbers can be reproduced with the estimator — the version used in this work is at commit `96875622c6b0e6f98a91ddeecaaa17b66dbc5a87`.

NIKEs. We caution the reader that all implementation details and numbers we present in this section are for Passive-SWOOSH only. To obtain a full picture of the performance of SWOOSH, the implementation will need to be augmented with a future implementation of the NIZK proof from [148]. As outlined in Section 5.5.3, the performance impact of adding the NIZK proof in terms of both size and computational effort depends on the concrete application scenario and may be negligible if key-generation performance is not critical and if NIZK proof verification can be outsourced to the PKI.

5.6.1 Implementation

Scheme (variant)	Security	Assumption	NI	PQ	Size in bytes		Cycles per op.	
					ct	*pk*		
CRYSTALS-Kyber (`Kyber-768` [191])	IND-CCA2	M-LWE	✗	✓	1 088	1 184	Gen	200 302
							Encaps	251 384
							Decaps	287 724
							SdK[35]	539 108
Classic McEliece (`mceliece348864` [7])	IND-CCA2	Binary Goppa Codes	✗	✓	96	261 120	Gen	46 715 060
							Encaps	31 000
							Decaps	112 178
							SdK[35]	143 178
ECDH (`X25519` [45])	HKR-CKS*[36]	CDH	✓	✗	—	32	Gen	28 187
							SdK	87 942
CTIDH (`CTIDH-1024` [29])	HKR-CKS*[37]	CSIDH	✓	✓	—	128	Gen	469 520 000
							SdK	511 190 000
This work (Passive-SWOOSH)	HKR-CKS	M-LWE	✓	✓	—	221 184	Gen	146 920 890
							SdK	10 612 666

Table 5.2: Comparison of select post-quantum KEMs and NIKEs. (NI: Non-Interactive; PQ: Post-Quantum)

As a NIKE, SWOOSH is composed of two major functions, the key generation procedure and the shared-key computation, the performance of which dictates the efficiency of SWOOSH.

In the case of the key generation, the matrix $\boldsymbol{A}$ is fixed and assumed to be in the NTT domain, so performance is dictated by the sampling of the secret and error vectors, as well as the computation of the public key which involves two NTT transformations, and a matrix multiplication followed by a polynomial addition. On Intel Skylake the cumulative execution time of the forward NTT accounts for ~ 10% of total key-generation time. This calculation is based on a single transformation taking 217 430 cycles, with a total of 64 transformations executed (32 for each secret and error vector). In contrast, the inverse NTT and noise-generation processes require 262 992 and 89 776 cycles, respectively. As for

[35] Encaps + Decaps
[36] See Section 5.6.2.
[37] See Section 5.6.2.

the shared-key computation, its performance is mainly dictated by the random offset computation, which requires the use of cSHAKE [132] and the polynomial base multiplication required to calculate $\boldsymbol{k}'$ (see Figure 5.3). Similar to other schemes, the shared-key derivation also performs rounding of the shared key, however its execution time is negligible. At a high level, the architecture of our implementation is divided into two distinct parts: low-level field arithmetic over $\mathbb{F}_q$ that is implemented using the Jasmin language [15, 18], and polynomial arithmetic in $\mathcal{R}_q$ as well as the scheme itself, both of which are implemented in Rust.

The structure largely mimics the abstract specification in Figure 5.3. The main difference is that, like other lattice-based schemes [11, 191], we encode and transmit public keys in NTT domain. This massively reduces the number of cycles required for shared-key computation. In addition, as discussed in Section 5.5.3, we assume that the role of each party is well defined and thus only compute one half of the key. Finally, we implement the noise sampling in a slightly different way than one might expect; we will discuss this later in this section.

Zooming in on the low-level field arithmetic, the operations on integers modulo $2^{214} - 255$ require multiple-precision integers since native 64-bit registers are not large enough to store a single field element. This arithmetic is implemented through libjbn [38], a Jasmin library that exposes (modular) big-integer arithmetic.

Polynomial Arithmetic. On top of this layer, operations in polynomial rings are implemented using Rust, in addition to other functions such as reconciliation, matrix and noise generation. Similar to other lattice-based schemes, one of the more critical (and easier) operations to optimise (from a performance perspective) is polynomial multiplication. The naive algorithm for multiplying two polynomials in $\mathcal{R}_q$, sometimes called Schoolbook multiplication, involves multiplying all pairs of coefficients, calculating their sum and reducing modulo $X^d + 1$. However, the complexity of this approach is quadratic in the number of coefficients and thus quite costly.

The *Number Theoretic Transform* (NTT) provides a more efficient approach for polynomial multiplication with quasi-logarithmic time complexity $\mathcal{O}(d \log(d))$ instead of $\mathcal{O}(d^2)$. For a detailed discussion on the NTT refer to [192].

As is the case for other implementations [191, 11], we use an in-place NTT which requires bit-reversal operations in the forward and inverse transforms but uses less memory. Another optimisation is to make the NTT a part of our scheme, which means that the matrix $\boldsymbol{A}$ is sampled in the NTT domain, and the secret and public keys are stored in the NTT domain. This results in the NTT only being used three times, once for the shared-key derivation and

[38] See `https://github.com/formosa-crypto/libjbn`.

twice in the key generation to transform the secret and error vectors, which are sampled in the normal domain, to the NTT domain before computing the public key. A common trick to speed-up the NTT transformation when using Montgomery reduction [163], as is the case for libjbn, is to use pre-computed constants in Montgomery form $\zeta \cdot R \pmod{q}$.

Noise Sampling and Matrix Generation. Both the matrix generation and noise-sampling procedures use a seed, either set as a system parameter for $\boldsymbol{A}$ or as a secret input to a PRG in the case of $\boldsymbol{s}$ and $\boldsymbol{e}$, to produce a stream bytes from which the distributions are sampled. In the case of matrix generation this is achieved via rejection sampling on the stream of bytes produced by an extendable output function (XOF). The noise-sampling procedure, used for generating the secret key and the error vector, samples these vectors from a centred binomial distribution using the output of a PRF with a random seed. As with other schemes where multiplication is optimised using the NTT, the choice of (symmetric) primitive that underlies these functions tends to be a deciding factor for the performance. We chose cSHAKE [132] based on Keccak [93] as the underlying primitive for the XOF and AES256-CTR for the PRF used in noise sampling.

Similar to the NewHope scheme [11], for efficiency reasons the secret and error vectors are sampled from a centred binomial distribution rather than a discrete Gaussian distribution. Using ternary noise means that each coefficient can be generated from only 2 bits and thus, the generation of a polynomial in $\mathcal{R}_q$ only requires $(32 \cdot 256 \cdot 2)/8 = 2048$ (pseudo-random) bytes. Intuitively, our CBD definition in Figure 5.3 when a and b are sourced from a PRG, maps 00_b and 11_b to $0 \mod q$ with 50% probability, 10_b to $1 \mod q$ and 01_b to $-1 \mod q$ with 25% probability each. Our implementation differs from the specification by applying signed reduction modulo 3 to each two bit block and converting it to a congruent value in $\mathbb{F}_q$, as opposed to using big-integer field arithmetic to map bits a and b to an element in $\mathbb{F}_q$. Although this approach produces a different mapping (11_b to $-1 \mod q$, 00_b and 10_b to $0 \mod q$ and 01_b to $1 \mod q$), the distribution of the outputs is identical. Due to the size of our field elements, this approach results in a considerable speed up in the noise sampling. The random offset used in our scheme is generated by performing rejection sampling on the output of cSHAKE-256 [132].

5.6.2 Performance and Security Evaluation

In this section we evaluate the performance of our scheme and compare it to other NIKEs and KEMs. Additionally, we present a comparison of key sizes and the properties of each scheme such as post-quantum security, and whether they are non-interactive, and the security notions they satisfy.

Security Models. To set a common ground for comparison, we note that the SdK algorithm in a NIKE can be viewed as simultaneously running both Encaps and Decaps of a KEM. KEMs should satisfy the standard security notions of IND-CCA security, with schemes such as CRYSTALS-Kyber [191] satisfying indistinguishability under *adaptive* chosen ciphertext attacks (IND-CCA2 security). On the other hand, we implemented the passively-secure version of our NIKE. We argue that this comparison is still meaningful because, in the context of a Public Key Infrastructure (PKI), NIZK verification can be performed once and for all by the PKI. Therefore, the runtime of our scheme is primarily influenced by the operations of the passively-secure NIKE.

We also note that the *"unhashed"* version of ECDH (`X25519` [45]), for which we report benchmarks, is insecure in the (DKR-)CKS model. In an unhashed NIKE where shared keys are not linked to specific identities, an adversary, A, can carry out a two-step attack: (i) A registers the public key g^a belonging to an honest party, Alice, as if it were their own key ($\texttt{RegCorUsr}(\texttt{ID} = \mathsf{A}, pk = g^a)$). (ii) Next, A requests the shared key between themselves and another honest party, say Bob, who possesses a public key g^b ($\texttt{RevCorQue}(\texttt{ID} = \mathit{Bob}, \texttt{ID} = \mathsf{A}) \rightarrow g^{ab}$). This immediately gives the shared key g^{ab} between Alice and Bob. Due to the homomorphic properties of (EC)DH, this attack remains effective (requiring only a minor modification) even if A is prohibited from registering keys of existing users. On the other hand, a hashed version of (EC)DH can be shown to be DKR-CKS secure in the ROM under the *Strong Computational Diffie-Hellman* assumption.

Although there is no proof of CTIDH (`CTIDH-1024` [29]) in the CKS model, the unhashed variant (excluding identities) of CTIDH faces similar security issues for the same reason. Recent work [92] proved that achieving actively secure NIKE from CSIDH inherently depends on *"very strong variants of the Group Action Strong CDH"* problem.

Nevertheless, it is important to note that Passive-SWOOSH is indeed strictly weaker than unhashed DH, as a key-recovery attack becomes possible if the adversary is allowed to register corrupted keys. Furthermore, from a practical point of view, enhancing the security of ECDH and CTIDH does not come with a significant increase in costs, as the computation time for hashing is still minimal, compared to the dominating modular-arithmetic cycle counts.

Benchmarks. The benchmark results for Passive-SWOOSH were obtained on an Intel Core i7-6500U (Skylake) running on a single core with Hyper-threading and TurboBoost disabled. The Rust compiler version used for the benchmarks was 1.62.1[39] and the Jasmin compiler version was 2022.09.0. We report the median cycle counts of 10000 runs. In Table 5.2 we list the results and compare to the cycle counts of Kyber-768 (on the same hardware), mceliece348864 as

[39]The following build configuration options/values were used: `opt-level=3` and `target-cpu="native"`.

reported in [46], CTIDH-1024 as reported in [29, Sec. 8], and of lib25519 [164], on Intel Skylake CPUs.

As expected, the pre-quantum X25519 [45] scheme is orders of magnitude faster than Passive-Swoosh for key generation. However, in many applications of NIKEs, keys are re-used many times and what is more critical is the performance of shared-key computation. Here the gap to pre-quantum X25519 is considerably smaller and Passive-Swoosh outperforms the only real post-quantum competitor CTIDH by a factor of 48. Compared to post-quantum KEMs, such as Kyber-768 and mceliece348864, the performance gap increases to several orders of magnitude, similarly to X25519. However, these schemes require further rounds of interaction, which not only presents additional overhead, but also means they cannot be used as drop-in replacements for a NIKE.

Additionally, as shown in Table 5.2, CTIDH, Kyber, and X25519 have a public key size several orders of magnitude smaller than Passive-Swoosh. In this aspect, only Classic McEliece has a public key size comparable to that of Passive-Swoosh, even when taking into account the expected size of the proof of knowledge (see Section 5.5.3).

5.7 Conclusions

In this work, we constructed a NIKE based on the M-LWE problem, with a proof in the QROM. Our scheme is based on the standard blueprint, but with an additional twist to guarantee provable security for arbitrary rings. Our optimised implementation shows that our scheme offers reasonable computational performance and key sizes that should be acceptable for most applications. We view our work as the first evidence *contradicting* the folklore belief that lattice-based NIKE is too inefficient to be used in practice. As future work, we plan to explore applications of our scheme to more complex protocols and to formally verify the correctness of (parts of) our implementation.

6

Jazzline: Composable CryptoLine functional-correctness proofs for Jasmin programs

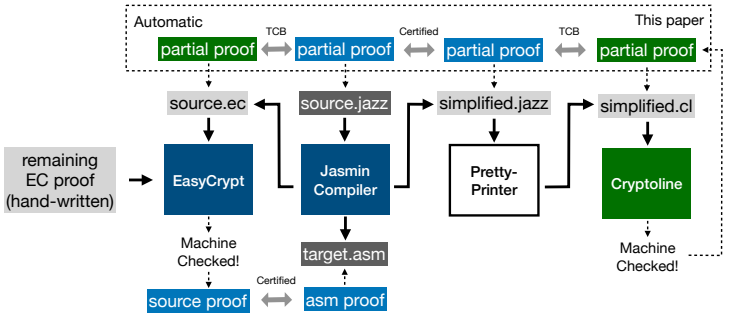
This chapter is for all practical purposes identical to the paper *"Jazzline: Composable CryptoLine Functional Correctness Proofs for Jasmin Programs"* [16], co-authored with José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Lionel Blatter, Gustavo Delerue, João Diogo Duarte, Benjamin Gregoire, Tiago Oliveira, Pierre-Yves Strub, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Published at ACM CCS 2025.

Abstract. Jasmin is a programming language for high-speed and high-assurance cryptography. Correctness proofs of Jasmin programs are typically carried out deductively in EasyCrypt. This allows generality, modularity and composable reasoning, but does not scale well for low-level architecture-specific routines. CryptoLine offers a semi-automatic approach to formally verify algebraically-rich low-level cryptographic routines. CryptoLine proofs are self-contained: they are not integrated into higher-level formal verification developments. This paper shows how to soundly use CryptoLine to discharge subgoals in functional-correctness proofs for complex Jasmin programs. We extend Jasmin with annotations and provide an automatic translation into a CryptoLine model, where most complex transformations are certified. We also formalize and implement the automatic extraction of the semantics of a CryptoLine proof to EasyCrypt. Our motivating use-case is the X-Wing hybrid KEM, for which we present the first formally verified implementation.

Keywords: Computer-Aided Cryptography, Formal Verification, EasyCrypt, Jasmin, CryptoLine

6.1 Introduction

Cryptographic libraries are a cornerstone of computer security. As such, it is essential that these libraries are correctly implemented. Unfortunately,



(a) Diagram of the extended tool-chain.

Algorithm expandDecapKey(sk):
$(\mathsf{coins}_M, sk_X) \leftarrow \mathsf{SHAKE256}(sk, 96)$
$(pk_M, sk_M) \leftarrow \mathsf{ML\text{-}KEM\text{-}768.KeyGen}(\,;\mathsf{coins}_M)$
$pk_X \leftarrow \mathsf{X25519}(sk_X, \mathsf{X25519_BASE})$
Return (sk_M, sk_X, pk_M, pk_X)

Algorithm KeyGen( ):
$sk \xleftarrow{\$} \{0,1\}^{256}$
$(sk_M, sk_X, pk_M, pk_X) \leftarrow \mathsf{expandDecapKey}(sk)$
Return $((pk_M, pk_X), sk)$

Algorithm Encapsulate((pk_M, pk_X)):
$ek_X \xleftarrow{\$} \{0,1\}^{256}$
$cph_X \leftarrow \mathsf{X25519}(ek_X, \mathsf{X25519_BASE})$
$shk_X \leftarrow \mathsf{X25519}(ek_X, pk_X)$
$(shk_M, cph_M) \leftarrow \mathsf{ML\text{-}KEM\text{-}768.Encap}(pk_M)$
$shk \leftarrow \mathsf{SHA3\text{-}256}(shk_M, shk_X, cph_X, pk_X, \mathsf{label}_X)$
Return $(shk, (cph_M, cph_X))$

Algorithm Decapsulate($sk, (cph_M, cph_X)$):
$(sk_M, sk_X, pk_M, pk_X) \leftarrow \mathsf{expandDecapKey}(sk)$
$shk_M \leftarrow \mathsf{ML\text{-}KEM\text{-}768.Decap}(cph_M, sk_M)$
$shk_X \leftarrow \mathsf{X25519}(sk_X, cph_X)$
$shk \leftarrow \mathsf{SHA3\text{-}256}(shk_M, shk_X, cph_X, pk_X, \mathsf{label}_X)$
Return shk

(b) The X-Wing Hybrid KEM

guaranteeing functional correctness of cryptographic implementations remains extremely challenging. Therefore, there is a growing emphasis on using program verification for proving correctness of these implementations [35]. At one end of the spectrum, automated verification tools such as CryptoLine [199, 99] have been used to verify challenging and highly optimized subroutines, including multi-precision arithmetic, elliptic curve cryptography, and core components in lattice-based cryptography such as the Number Theoretic Transform. However, these approaches are limited to specific classes of programs—informally, straight-line code with statically resolved memory accesses, and are not compositional. At the other end of the spectrum, interactive verification tools such as EasyCrypt have been used to verify full algorithms, including high-speed implementation of the SHA-3 standard [17] and the recent FIPS-203 ML-KEM standard [129, 13]. However, these approaches are labour intensive and do not scale well for low-level subroutines. The goal of this work is to integrate both approaches in a single verification framework.

Contributions

This paper develops sound foundations and tooling for combining deductive and automated program verification, namely EasyCrypt and CryptoLine, and shows how the resulting framework helps verifying cryptographic implementations written in the Jasmin language. Our approach is based on the following steps and illustrated by a diagram in Figure 6.1a. The traditional proof development architecture for a Jasmin source program is shown on the left of the diagram. A Jasmin source file is compiled to assembly by the certified Jasmin compiler, which means that functional correctness established at the source will also hold at the assembly level. In parallel, a model of the source program is extracted to the EasyCrypt theorem prover, where the functional correctness proof for the source code is machine-checked.

In this paper we extend the Jasmin language and compiler to allow integration of CryptoLine in the tool-chain:

- We extend the Jasmin programming language with assume and assert statements via annotations in the source code. These statements allow users to state functional correctness contracts using an extension of the Jasmin language of expressions, which is general enough to write the annotations required for a CryptoLine-based correctness proof. Furthermore, our annotation language is higher-level than that used typically in CryptoLine proofs, as it includes big operators for summation and universal quantification, and so the resulting contracts are more convenient to work with in high-level proofs.

- We implement new verified compilation steps in the Jasmin compiler (these are specific to the new proof methodology we introduce here) that perform a Jasmin-to-Jasmin transformation that we call simplification. The resulting simplified Jasmin programs are equivalent to the source, but they are stripped of all high-level constructs, including branching, loops, arrays, etc., and they are also in static single-assignment form. However, we retain function calls to allow for a modular proof relying on CryptoLine. Furthermore, and crucially for this work, the compiler is proved to preserve the semantics of annotations.

- We state and prove a soundness meta-theorem that justifies the methodology illustrated in Figure 6.1a. The annotated Jasmin source program is simplified and then pretty-printed to CryptoLine. If CryptoLine accepts the program, then this yields a partial correctness result. Here the proof is partial for two potential reasons: 1) the possible use of assume statements in the proof and 2) the fact that verification in CryptoLine is done independently for each function in the call tree. Our meta-theorem then shows that the certified simplification of Jasmin programs permits lifting the CryptoLine proof back to the source Jasmin program.

- We extend the extraction to EasyCrypt, so that it includes the partial correctness proved by CryptoLine as axioms. Again, the way in which the

extraction to EasyCrypt is performed is justified by our meta-theorem. As a result, the effort of constructing a full EasyCrypt proof for the source Jasmin program is much reduced. In particular, we can synthethize the EasyCrypt proof that justifies our modular verification in CryptoLine of each function in the call tree. This means that the only non-trivial goals that remain to prove in EasyCrypt concern justifying the assumptions made when carrying out the CryptoLine proof.

The benefits of our approach are numerous, and they affect both the users of Jasmin and the users of CryptoLine:

- Functional correctness proofs for complex Jasmin programs can now be factored into high-level compositional reasoning—which is natural in EasyCrypt and typically carries over across implementations for different architectures—and automatic proofs for low-level functions in which CryptoLine excels.

- This level of automation still retains the strong end-to-end guarantees of EasyCrypt proofs: the program has a well defined semantics for all valid inputs and is correct with the input/output behavior given by the pre- and post-condition. We achieve this by making sure that all assumptions made in CryptoLine proofs, which are the responsibility of the user, need to be justified.

- The way in which we carry out CryptoLine proofs permits composing them using EasyCrypt. This is necessary for our end-goal of integrating the CryptoLine results in more complex proofs, but it also has implications for CryptoLine users. For example, one can use the approach proposed in this paper to carry out different proofs in CryptoLine for the same function, corresponding to different cases/pre-conditions, and then use EasyCrypt to combine these proofs into a single one that aggregates the various cases. Similarly, EasyCrypt can be used to justify the modular reasoning about various functions in a call tree.

- Effort can be shared and reused across different tools and teams.

We illustrate our approach with two examples. The first and main motivating example is X-Wing, where we use previous CryptoLine proofs of X25519 subroutines and compose them with previous EasyCrypt proofs of ML-KEM and SHA3. The second example is ML-KEM NTT, where we illustrate modular reasoning where parts of the call tree are justified in CryptoLine and other parts in EasyCrypt. Furthermore, we also show how we can reuse existing CryptoLine proofs for the NTT by bridging in EasyCrypt the algebraic description of the NTT (based on the Chinese-Remainder Theorem) used in CryptoLine contracts to the DFT-based description of the NTT used in the EasyCrypt proof of correctness wrt the ML-KEM standard.

Trusted Code Base

The TCB includes, of course, Coq, EasyCrypt and CryptoLine (where the latter two include SMT and CAS solvers). Furthermore, it includes the non-certified parts of the Jasmin compiler code, including parsing, typing and the safety checker, as well as extraction to EasyCrypt. This is inherent to the use of all the tools above. Specific to our work, the TCB is extended with a new form of EasyCrypt extraction for Jasmin programs that is aware of annotations and their semantics. Furthermore, we also create a new pretty-printing from Jasmin to CryptoLine. Both of these mechanisms are assumed to satisfy the properties required by our formal framework and Theorem 6 in particular.

Everything that we have not listed above in the TCB is machine-checked either in Coq or in EasyCrypt, as follows. The main extensions to the Jasmin formalization in Coq are: syntax, semantics, and compilation of annotations; verified compilation from Jasmin to "simplified Jasmin" (straight-line, array-free, etc.), which corresponds to Theorem 1; a new theorem (corresponding to Theorem 2) establishing reflection of traces from target to source. A new EasyCrypt library, proved once and for all, includes the following general results: definitions of trace types and trace validity; relations between notions of validity (Theorem 3). For each proof, the extraction to EasyCrypt, generates lemmas that can be easily discharged to guarantee that no mistakes are made in applying Theorems 4 and 5: proving that the program is well annotated for assumes (hypothesis 3 in Theorem 5); proving that the program is well annotated and therefore functionally correct (Theorems 4 and 5 assuming safety has been checked, and assuming that hypothesis 2 in Theorem 5 has been checked by CryptoLine).

Related work.

There is a large body of work that uses program verification for proving correctness of cryptographic code [35]. A majority of this work, including [208, 207, 23] and prior work on Jasmin, follows the classic approach of deductive verification. The design of CryptoLine [199] features automation over scalability and compositionality, so in this work we explore the possibility of combining two verification frameworks in a sound way. An alternative approach would have been to extend EasyCrypt directly with the ability to carry out CryptoLine-style proofs, which is the spirit of the work done in [207] for the Dafny framework. Another traditional approach is to generate correct programs from specifications. This is the approach followed by FIAT Crypto [94]; recently the approach has been extended with mechanisms to synthesize correct and optimized implementations [139]. This approach is appealing but is not applicable to the complex implementations that we consider.

Structure of the paper

Section 6.2 provides further motivation via an example. Sections 6.3 to 6.5 justify the theoretical soundness of our approach and the extensions to the Jasmin compiler. Sections 6.6 and 6.7 describe our implementation of the interaction with CryptoLine and EasyCrypt, respectively. Finally, Section 6.8 describes the examples and Section 6.9 adds some concluding remarks.

6.2 Motivating Example

Introducing X-Wing.

To motivate our approach we will use X-Wing [36, 79], a recently proposed hybrid Key Encapsulation Mechanism (KEM). X-Wing uses ML-KEM as a blackbox. ML-KEM was recently standardized by NIST as FIPS-203 [167] (Module-Lattice-Based Key-Encapsulation Mechanism). The goal of the X-Wing design is to provide a fallback to the security of ML-KEM, and guarantee at least classical security in the unlikely scenario that a weakness is uncovered in the post-quantum standard. To this end, X-Wing runs an elliptic-curve-based KEM-like construction in parallel with ML-KEM, so that the X-Wing shared key is derived using contributions from both the classical and the post-quantum constructions. This classical layer is based on the X25519 elliptic curve [143], and the overall design is optimized for performance: the final key is derived by passing to the final SHA-3 based key derivation function just enough information to guarantee security, while requiring only a single block absorption for the hash computation. The security analysis of X-Wing [36] shows that the construction is secure against post-quantum adversaries if ML-KEM is secure, and that it is secure against classical adversaries if a variant of the computational Diffie-Hellman problem holds in the X25519 group.

The X-Wing construction is given in pseudocode in Figure 6.1b.[40] Our goal is to obtain a high-speed and high-assurance implementation of X-Wing with minimal effort, which means reusing as much as possible pre-existing implementations and functional-correctness proofs. Looking at the design of the construction, there are three components that need consideration: ML-KEM-768, X25519, and two functions in the SHA3 family. Our starting point are the pre-existing formally verified implementations of ML-KEM [129, 13] and SHA3 [17]. These implementations are written in the Jasmin programming language for x86-64, and the corresponding functional-correctness proofs are written in the EasyCrypt proof assistant [37]. However, the functional correctness of the existing Jasmin implementations of X25519 were not yet formally verified.

[40] In this paper we follow the specification given in the Internet Draft [36, 79]

```
for i = 0 to 3 {
  ( hi, lo )   = #MULX ( _38,  r[i] );
  of, h[i]     = #ADOX ( h[i], lo, of );
  cf, h[i+1]   = #ADCX ( h[i+1], hi, cf );
}
```

```
mulx   %r12,%rax,%rbx
adcx   %rax,%r8
adox   %rbx,%r9
mulx   %r13,%rax,%rbx
adcx   %rax,%r9
adox   %rbx,%r10
mulx   %r14,%rax,%rbx
adcx   %rax,%r10
adox   %rbx,%r11
mulx   %r15,%rax,%r12
adcx   %rax,%r11
adox   %rdi,%r12
adcx   %rdi,%r12
mov    0x10(%rsp),%rdi
imul   %rdx,%r12
add    %r12,%r8
adc    $0x0,%r9
adc    $0x0,%r10
adc    $0x0,%r11
sbb    %rax,%rax
and    $0x26,%rax
add    %rax,%r8
```

```
( r[0], lo ) = #MULX ( _38, r[3] );
of, h[3]     = #ADOX ( h[3], lo, of );
cf, r[0]     = #ADCX ( r[0], z, cf );
of, r[0]     = #ADOX ( r[0], z, of );

_,_,_,_,lo = #IMULri ( r[0], 38 );
cf, h[0] += lo;
cf, h[1] += z + cf;
cf, h[2] += z + cf;
cf, h[3] += z + cf;
_, z -= z - cf;
z &= 38;
h[0] += z;
```

```
mull rbx rax rdx r12;
adcs carry r8 r8 rax carry;
adcs overflow r9 r9 rbx overflow;
mull rbx rax rdx r13;
adcs carry r9 r9 rax carry;
adcs overflow r10 r10 rbx overflow;
mull rbx rax rdx r14;
adcs carry r10 r10 rax carry;
adcs overflow r11 r11 rbx overflow;
mull r12 rax rdx r15;
adcs carry r11 r11 rax carry;
adcs overflow r12 r12 rdi overflow;
adcs carry r12 r12 rdi carry;
mull dontcare r12 rdx r12;
assert true && and [dontcare = 0@64,carry=0@1,overflow=0@1];
assume and [dontcare = 0,carry=0,overflow=0] && true;
adds carry r8 r8 r12;
adcs carry r9 r9 0x0@uint64 carry;
adcs carry r10 r10 0x0@uint64 carry;
adcs carry r11 r11 0x0@uint64 carry;
ghost carryo@bit: carryo = carry && carryo = carry;
sbbs carry rax rax rax carry;
assert true && carry = carryo;
assume carry = carryo && true;
mov overflow carry;
not zero@bit carry;
and rax@uint64 rax 0x26@uint64;
assert true && or [ and [carry=0@1, rax=0@64], and [carry=1@1, rax=0x26@64]];
assume rax = carry*0x26 && true;
adds carry r8 r8 rax;
assert true && carry=0@1;
assume carry=0 && true;
{ eqmod (limbs 64 [r8, r9, r10, r11]) (limbs 64 [a0, a1, a2, a3] * limbs 64 [a0, a1, a2, a3])
    ((2**255)-19)   &&  true }
```

Figure 6.2: Snippets of three implementations of the same X25519 basic subroutine, which computes the square of a 32-byte integer represented as 4x64-bit registers modulo $2^{255} - 19$. Top Left: Jasmin code. Top Right: openssl assembly. Bottom: CryptoLine proof script for openssl assembly.

X25519 implementations

Implementations of X25519 are pervasive in cryptographic libraries, and they are typically structured in two layers: low-level leaf functions that perform multi-precision arithmetic modulo prime $p = 2^{255}$-19 and encoding/decoding routines, whereas a higher level function performs the relevant scalar multiplication operation in the group of elliptic curve points. The top level interface for this operation, shown as $\mathsf{X25519}(\cdot,\cdot)$ in Figure 6.1b, takes a 32-byte scalar u and a 32-byte representation of an elliptic curve point P, and returns the 32-byte representation of uP.

The challenging aspect of proving the correctness of an X25519 scalar multiplication operation resides in the multi-precision computations performed in the leaf functions. Indeed, in a tool such as EasyCrypt, once the leaf functions have been proved to perform the correct low-level computations, it is relatively straightforward to argue that the top-level function is calling the low-level functions in the correct sequence using Hoare logic. Furthermore, this high-level proof is essentially the same for any target architecture and multi-precision representation of finite-field elements. On the other hand, proving the correctness of the multi-precision arithmetic implementation requires significant effort in an interactive proof, which is hard to amortize across implementations for different target architectures and choices of finite field element representations. Indeed, for such low level functions, searching for an automated proof methodology is well justified.

Pre-existing CryptoLine proof

In this work we consider CryptoLine [199, 99], a tool that allows precisely such an automated proof methodology. CryptoLine has been specifically designed for the verification of low-level implementations of mathematical constructs, including multi-precision arithmetic. Moreover, the CryptoLine set of examples includes proofs of the openssl implementation of X25519 finite field operations in x86-64 assembly, which are very close to the Jasmin implementation of the same functions. To illustrate this similarity, Figure 6.2 shows snippets from three versions of a routine that performs squaring modulo 2^{255}-19. The first one, on the left, is the Jasmin implementation we are interested in formally verifying. In the middle we show assembly code taken from the openssl implementation of the same routine.[41] Finally, on the right, we can see part of the CryptoLine proof script for the openssl assembly.[42] The snippets show part of the squaring routine that performs modular reduction from a 64-byte intermediate result to the 32-byte final result, i.e., it computes the modular reduction of $a^2 \pmod{2^{255}-19}$ where $a \in [0; 2^{256})$ and is represented as four (packed) 64-bit words. It is clear that the Jasmin and openssl code are doing essentially the same thing, and this pattern

[41] https://github.com/openssl/openssl/blob/master/crypto/ec/asm/x25519-x86_64.pl

[42] https://github.com/fmlab-iis/cryptoline/blob/master/examples/openssl/x25519/x86_64/x25519_fe64_sqr.cl

repeats itself with minor differences in scheduling and instruction selection for the entire implementation of the X25519 leaf function. Our expectation is therefore that the CryptoLine proof actually applies to the Jasmin code, and our motivation in this paper is to allow reusing such proofs in a sound way.

Let us now take a closer look at the snippet of the CryptoLine proof script shown in Figure 6.2 (right). At the end of the code, one can see the post-condition. Like all predicates over CryptoLine program states, it has two parts $Q_A \,\&\&\, Q_R$, where Q_A is called an algebraic predicate and Q_R is called a range predicate. Algebraic predicates refer to the (in this case, unsigned) integer interpretation of the values of program variables, whereas range predicates refer to program variables as fixed-length bit vectors.

In this example, Q_R is the trivial true post-condition, and the algebraic post-condition Q_A is the congruence $a^2 \equiv r \pmod{2^{255} - 19}$, where a is the integer represented by inputs to the function (a_0, a_1, a_2, a_3), and r is the integer represented by the function outputs $(r_8, r_9, r_{10}, r_{11})$. Here, all inputs and outputs are 64-bit values interpreted as unsigned integers, as the representation uses packed 64-bit limbs, s.t. for example $a := \sum_{i=0}^{3} 2^{64i} a_i$.

The CryptoLine proof consists of two parts: a model of the program, consisting of a sequence of CryptoLine instructions, and a set of annotations that introduce ghost variables and insert assert and assume statements.[43] The use of assert and assume statements in this proof is illustrative of an important point about CryptoLine proofs. Assert statements (in this case, of range expressions) are required by CryptoLine to hold in all relevant executions. On the other hand, assume statements can be used to constrain the set of executions for which the proof is required to hold (i.e., possibly beyond the restriction imposed by the pre-condition): an algebraic assumption such as those introduced in the current proof is taken as an additional equality that can be assumed to hold over the integers when the program is being checked for correctness. A closer look shows that algebraic assume statements in Figure 6.2 (bottom) are actually implied by the range assert statements inserted immediately before them, but this is not checked by the tool and is left to the user to confirm by inspection. This approach is extremely powerful, as it permits using the power of the SMT solver backends (used to discharge) range predicates to simplify the algebraic model (e.g., by incorporating an equation that sets a carry variable to 0) and can be critical to enable an automatic proof based on a Computer Algebra System (CAS).

Our goal of using CryptoLine to prove properties of Jasmin programs therefore raises three questions:

1. How to construct a CryptoLine model of a Jasmin program?

2. How to annotate the CryptoLine model of the Jasmin program?

3. How to integrate the CryptoLine proof in higher-level proofs.

[43] For a more detailed explanation of the CryptoLine semantics, please refer to Section C.1

```
param bool sub = false; param int MAX = 100; param int LEN = 2;
abstract predicate int u16i(u16);

fn source(reg ptr u16[LEN] a b) -> reg ptr u16[LEN]
requires {\all (i \in 0:LEN) (0 <= a[i] && a[i] < MAX)}
requires {\all (i \in 0:LEN) (0 <= b[i] && b[i] <= a[i])}
ensures {\all (i \in 0:LEN) (0 <= u16i(result.0[i])  &&
                             u16i(result.0[i]) <= 2*MAX)}
{ for i = 0 to LEN { if sub { a[i] -= b[i]; } else   { a[i] += b[i]; } assert
    (0 <= a[i]); }
  return a; }
```

```
fn simplified (reg u16 a_0_id_151, reg u16 a_1_id_152,
               reg u16 b_0_id_153, reg u16 b_1_id_154) -> (reg u16, reg u16)
requires {(((((16u) 0) <=u a_0_id_151) && (a_0_id_151 <u ((16u) 100))) &&
              ((((16u) 0) <=u a_1_id_152) && (a_1_id_152 <u ((16u) 100))))}
requires {(((((16u) 0) <=u b_0_id_153) && (b_0_id_153 <=u a_0_id_151)) &&
              ((((16u) 0) <=u b_1_id_154) && (b_1_id_154 <=u a_1_id_152)))}
ensures {((((0 <= u16i(a_0_id_155)) && (u16i(a_0_id_155) <= 200)) &&
              ((0 <= u16i(a_1_id_156)) && (u16i(a_1_id_156) <= 200)))}
{reg u16 a_0_id_155; reg u16 a_1_id_156;
  (_, _, _, _, _,a_0_id_155) = #ADD_16(a_0_id_151, b_0_id_153);
  assert (((16u) 0) <=u a_0_id_155);
  (_, _, _, _, _,a_1_id_156) = #ADD_16(a_1_id_152, b_1_id_154);
  assert (((16u) 0) <=u a_1_id_156);
  return (a_0_id_155, a_1_id_156); }
```

Figure 6.3: Example of annotated source (top) and simplified Jasmin (bottom) programs.

We now address each of these questions in turn.

Constructing the CryptoLine model

CryptoLine models must satisfy a set of strict requirements, such as being loop and branch free, to be correctly typed wrt to bit vector sizes and variable signedness, and to satisfy a notion of safety that, intuitively, means that there are no integer overflows in any of the bit vector computations.

The typical CryptoLine approach to creating a model is semi-automated, but it is not formally verified. One first creates a .gas file using a debugging tool, which is essentially a trace of the execution of assembly instructions where all operands have been resolved to either registers or constants (including memory addresses). This sequence is straight-line, and it provides enough information that permits rewriting the .gas file (almost 1-line-to-1-line) using an automatic script to obtain the model. This rewriting script manually is created for each proof, and is part of the proof process.

The above CryptoLine methodology has the advantage of being source-language-agnostic. However, for our purposes, the source language is fixed to Jasmin, and Jasmin is supported by a formally verified compiler. We therefore propose an alternative CryptoLine model-generation process where we extend the Jasmin compiler, and its proof, to establish a formal connection between Jasmin programs and their corresponding CryptoLine models and proofs. To this end, we created in the Jasmin compiler a new compilation path that can convert a restricted class of Jasmin programs—which may include high-level control-flow, such as unrollable for loops, integer variables, arrays of registers and stack positions, etc.—into a lower level format that can be pretty-printed as a CryptoLine program. In what follows, we refer to the Jasmin program we want to prove correct as *source* and to the transformed program that is pretty-printed to CryptoLine as *simplified*. Figure 6.3 shows a simple example of the source and simplified Jasmin programs. The source program uses arrays and for loops. The resulting target simplified program is very close to a CryptoLine program, in that all high-level features were removed: the program is straight-line, it uses no arrays, it is static single-assignment. One goal of our approach is to minimize human intervention in constructing the CryptoLine model and proof. We therefore fix two default models of Jasmin programs (for computations over signed and unsigned integers) and require user intervention only when it is necessary to locally change the modeling of specific instructions via translation directives inserted at the source level. The details are given in Section 6.6. Furthermore, as we discuss next, the annotations required for the proof can be added directly to the source Jasmin program.

Annotating Jasmin programs

We extended the Jasmin compiler with support for an extensible annotation language that supports preconditions, post-conditions, as well as assertions that refer to the program state during evaluation. Figure 6.3 shows also examples of these annotations in the source program, and how the compiler translates them into the simplified program. Note, in particular the use of high-level big operators such as all to have a compact syntax for the predicates over the program state, and how these are removed in the annotations of the simplified program. In the following sections we will give the formal details about why this means that we can perform a proof over a simplified program, and still have the proof apply to the Jasmin source we are interested in. Here we just give the general intuition. In short, the compiler proof has been enriched to cover an extended version of the semantics that is *annotation aware*: the value of all assertions are registered by the semantics in a trace during program evaluation. Moreover, this proof of correctness provides a stronger semantic *equivalence* relation between source and simplified programs that allows reasoning about correctness in either of the programs interchangeably.

EasyCrypt semantics of a CryptoLine proof

Using the above extensions to the Jasmin compiler, we can already construct CryptoLine proofs of Jasmin programs. However, our goal is to be able to use such results in larger proof developments. For example, in this paper, while we use CryptoLine to prove correctness of leaf functions that carry out X25519 multi-precision arithmetic, we use these results in a larger proof of correctness for the full X25519 implementation, which in turn is integrated into the proof of correctness of the X-Wing hybrid KEM construction in Jasmin.

Our final contribution is a new extraction of annotated Jasmin programs to EasyCrypt that exports the semantics of a CryptoLine proof in a format that can be composed with other proofs. Specifically, we formalize the meaning of a CryptoLine proof in terms of the Jasmin semantics, and create in EasyCrypt an exact description of what CryptoLine has proved in terms of these semantics. This description leaves as proof goals to be discharged interactively in EasyCrypt any unproved assumptions made in the CryptoLine proofs, thereby enabling a flexible splitting of proof effort between the two tools. For the X-Wing example, the proof goals that arise in EasyCrypt to complete the CryptoLine proofs are the justification of a common technique in CryptoLine: asserting under bit-string semantics that a bit is 0, and then assuming an algebraic predicate that is implied by this fact over the integers. Discharging these goals in EasyCrypt is simple, and can be seen as machine-checking what the human user was already able to check by direct inspection. The examples we present at the end of the paper illustrate additional ways in which we can modularize a proof to better take advantage of the strengths of interactive and automatic proof techniques.

6.3 Annotated Jasmin Programs

We have extended the Jasmin syntax and semantics to allow for two types of annotations. Function declarations can be annotated with pre- and post-conditions using the keywords requires and ensures, and *annot* statements can be added anywhere in the code. All annot statements, $annot(k, e)$, have an associated kind assert or assume and a boolean expression. For simplicity, we write assert e for $annot(\mathsf{assert}, e)$ (resp. for assume). We have also extended the syntax of Jasmin expressions with a generic (high order) fold operator over integers, which allows to easily encode big operators like all and sum, and we also allow the user to extend the language with abstract type and abstract operators. These operators can only be used in annotations, since they have no semantics and the compiler does not know how to compile them to assembly.

Semantics of an annotated program

To account for annotations, the semantics of Jasmin programs have been enriched in the Coq development. The evaluation of a Jasmin program defines, not only a

$$\frac{[\![p(f)_{pre}]\!](\mathbf{v_a}) = \mathsf{b} \qquad p(f)_c, (\varnothing[p(f)_{param} := \mathbf{v_a}], [(\mathsf{assume}, b)]) \Downarrow^* (s', t) \qquad [\![p(f)_{res}]\!](s') = \mathbf{v_r} \qquad [\![p(f)_{post}]\!](\mathbf{v_a}, \mathbf{v_r}) = \mathsf{b'}}{f, \mathbf{v_a} \Downarrow^p_{call} \mathbf{v_r}, t +\!\!+ [(\mathsf{assert}, b')]}$$

$$\frac{a, s \Downarrow \mathbf{v_a} \qquad [\![p(f)_{pre}]\!](\mathbf{v_a}) = \mathsf{b} \qquad f, \mathbf{v_a}, \Downarrow^p_{call} \mathbf{v_r}, t \qquad [\![p(f)_{post}]\!](\mathbf{v_a}, \mathbf{v_r}) = \mathsf{b'} \qquad t'' = t +\!\!+ [(\mathsf{assert}, b)] +\!\!+ t' +\!\!+ [(\mathsf{assume}, b')]}{(r = f(a), (s, t)) \Downarrow ((s[r := \mathbf{v_r}]), t''}$$

Figure 6.4: Semantics of call in Jasmin with annotations

relation between an initial state and a final state, but also the construction of a *trace* t keeping track of the annotations evaluated during the program execution. The trace is simply a list of pairs kind/predicate value. Formally, an *annot* statement leaves the state of the program unchanged, but it updates the trace with the kind and Boolean value of the associated expression in the current state.

$$\frac{[\![e]\!](s) = \mathsf{b}}{annot(k, e), (s, t) \Downarrow (s, t +\!\!+ [(k, b)])}$$

The semantics of a function call are also modified as shown in Figure 6.4. They add to the trace the kinds and values of the pre- and post-condition in two steps: the caller asserts the pre-condition before entering the callee, and assumes the post-condition upon return. The callee assumes the precondition upon entry, and asserts the post-condition before returning.

Finally, in what follows we will restrict our attention to *safe* Jasmin programs, which we define as follows.

Definition 1 (Safe program). *We say a Jasmin program c is safe when* $\forall s. \exists s', t. (c, (s, \epsilon)) \Downarrow^* (s', t)$.

6.4 Certified compilation with traces

Our interaction with CryptoLine requires a simplified Jasmin program, where all arrays, if statements and for loops have been removed in order to obtain a straight-line, static single-assignment program that can be pretty-printed to the CryptoLine language in a trivial way. To do so we have created inside the Jasmin compiler a new compilation pipeline that is specific to CryptoLine and that removes all high-level features from an input Jasmin program. We will call this the *CryptoLine version* of the Jasmin compiler and, when clear from the context, the *program simplification* performed by the Jasmin compiler. This pass rejects Jasmin programs that include features not supported by our CryptoLine backend, such as direct memory access and array accesses that

cannot be resolved at compile time. The simplification unrolls loops, replaces arrays with variables, and propagates constants (including those stored in global tables). Furthermore big operators in annotations are removed by partial evaluation. The final program is a valid annotated Jasmin program in static single-assignment form. The proof of the compiler (CryptoLine version) has been extended to take into account traces.

Theorem 1 (Preserving Trace). *The simplification performed by the Jasmin compiler preserves traces:*

$$\forall \sigma, s, t.\ (c, \sigma) \Downarrow^* (s, t) \implies \exists s'.\ (\mathcal{T}(c), \sigma) \Downarrow^* (s', t)$$

Where $\mathcal{T}(c)$ denotes the compilation of a program c.

A well known fact is that, because the semantics of Jasmin is deterministic, the above theorem implies the converse for safe programs.

Theorem 2. *The simplification performed by the Jasmin compiler reflects traces of safe programs.*

$$\forall \sigma, s, t.\ (\mathcal{T}(c), \sigma) \Downarrow^* (s, t) \implies \exists s'.\ (c, \sigma) \Downarrow^* (s', t)$$

Trace validity

In our theoretical framework we will also need the notions of trace and sub-trace validity.

Definition 2 (Valid Trace). *A trace t is valid, written $valid(t)$, when all the Boolean values in the trace are true (written $\top$).*

A CryptoLine proof does not guarantee full trace validity, but instead a restricted notion of sub-trace validity that we formalize in Section 6.6. Intuitively, CryptoLine guarantees that the assert subtrace is valid in the restricted set of executions where the assume trace holds.

Definition 3 (Valid Sub-trace). *We say the sub-trace of trace t of kind k is valid, written $valid(t)_{\mathsf{k}}$, when $valid(t)_{\mathsf{k}}$ is a derivation of the following rules:*

$$\frac{}{valid(\epsilon)_{\mathsf{k}}} \qquad \frac{valid(t)_{\mathsf{k}}}{valid((_, \top) :: t)_{\mathsf{k}}} \qquad \frac{k' \neq \mathsf{k}}{valid((k', \bot) :: _)_{\mathsf{k}}}$$

The intuition of this notion of validity is that, when checking for kind k, we are only interested in validity for executions in which annotations of the other kinds evaluate to true. When this is *not* the case, then validity trivially holds. In fact, this derivation is essentially encoding an implication: $valid(t)_{\mathsf{k}}$ holds if either some annotation of kind $k' \neq \mathsf{k}$ is false, or all annotations of kind k are true. An important property of sub-trace validity is completeness, which we prove in EasyCrypt for the concrete case of assume/assert.

Theorem 3 (Complete validity splitting).

$$valid(t)_{\mathsf{assert}} \wedge valid(t)_{\mathsf{assume}} \implies valid(t)$$

6.5 Methodology

So far we have discussed the meaning of an annotated program, so now we are ready to discuss our methodology to prove properties of Jasmin programs by relying on annotations.

To this end, we introduce two notions of correctness for Jasmin programs; one that permits establishing a weaker notion of partial correctness, and another one that permits reasoning about our target notion of total correctness.

Definition 4 (Well annotated program for a kind). *For given program c, and arbitrary predicates P and Q, we say a program is well annotated for kind* k *when:*

$$\{P\}c\{Q\}_{\mathsf{k}} \triangleq \\ \forall s, s', t. P(s) \implies (c; \mathsf{assert}\ Q, (s, \epsilon)) \Downarrow^* (s', t) \implies valid(t)_{\mathsf{k}}$$

We say a program is well annotated when the implication holds for $valid(t)$, i.e., execution always generates a full valid trace.

Intuitively, for the particular case of programs annotated only with assume and assert stements, a partially well annotated program is guaranteed to satisfy all assertions (including the post-condition) but only in executions where the assume annotations turn out to be true. Indeed, such a result says nothing about the executions in which the pre-condition and internal assume statements turn out to evaluate to false.

The stronger notion of well-annotated programs requires all annotations to be valid whenever the program starts from a state in which the precondition holds. Clearly, for safe programs, a well-annotated program is also totally correct in the standard sense:

Definition 5 (Total correctness).

$$[P]c[Q] \triangleq \forall s. P(s) \implies \exists s', t. (c, (s, \epsilon)) \Downarrow^* (s', t) \wedge Q(s')$$

Theorem 4. *A well annotated safe program is totally correct.*

The proof of this theorem is trivial, since reaching a final state is guaranteed by safety, and a well annotated program is known to satisfy the post-condition due to the assertion over the final state.

Methodology

Our methodology relies on the fact that we decompose a proof of functional correctness by 1) proving that a simplified version of the program is partially well annotated for asserts using CryptoLine (this first part is done fully automatically); and 2) proving that the source program is partially well annotated for assumes using EasyCrypt. This latter part is not always necessary,

but it allows us to flexibly split the proof effort between CryptoLine and EasyCrypt, which is useful in several cases as we will illustrate with our examples. Formally, the soundness of our approach is captured by the following theorem.

Theorem 5 (Soundness). *Let c be an annotated Jasmin program and let $\mathcal{T}(c)$ denote its simplified counterpart. Then following conditions imply that $[P]c[Q]$ holds:*

1. *c is safe*
2. $\{\mathcal{T}(P))\}\ \mathcal{T}(c)\ \{\mathcal{T}(Q))\}_{\mathsf{assert}}$
3. $\{P\}c\{Q\}_{\mathsf{assume}}$

Proof. By Theorem 2 we know that (2) implies $\{P\}\ c\ \{Q\}_{\mathsf{assert}}$. This, combined with (3) allows us to apply Theorem 3 and derive that the program is well annotated. Finally, we complete the proof by applying Theorem 4. ∎

6.6 Proving Partial Correctness using CryptoLine

We now explain how we use CryptoLine as a decision procedure for the partial correctness of simplified Jasmin programs.

CryptoLine and its semantics

CryptoLine programs are straight-line and static single-assignment, just like simplified Jasmin programs (see Appendix C.1 for more on CryptoLine). CryptoLine variables are also declared as words of fixed size, but they have a type annotation that indicates how they should be interpreted as an integer: they are either signed or unsigned. The semantics of CryptoLine instructions is defined by interpreting the values as integers, performing a computation over the integers, and injecting the result back into the type. The CryptoLine notion of safety requires proving that all such injection operations performed by the semantics are overflow-free (this depends on signedness). Otherwise the semantics raises an error. Like simplified Jasmin programs, CryptoLine programs are annotated with pre-conditions and post-conditions, and they can have intermediate assert and assume statements. The semantics of CryptoLine evaluates to an error whenever an assert statement evaluates to false. For assume statements there is no error: the execution of the program is blocked. The following notion of correctness, captures the meaning of successful CryptoLine proof [199, 99].

Definition 6 (CryptoLine correctness). *We say a program c is correct wrt CryptoLine for predicates P and Q if:*

$$\{P\}c\{Q\}_{\mathsf{CL}} \triangleq \forall s, s'.\ P(s) \implies (c, s) \Downarrow^{*}_{\mathsf{CL}} s' \implies s' \neq \mathsf{error} \wedge Q(s')$$

From simplified Jasmin to CryptoLine

We construct a pretty-printing function $\mathcal{P}$ from an annotated (simplified and safe) Jasmin program $c \coloneqq [c_1, \ldots, c_n, \mathsf{return}\ v]$ that, by construction, has the following properties.

- State equivalence: there is a one-to-one correspondence between the variables in the Jasmin program and the subset of variables in the pretty-printed program that can occur in pretty-printed CryptoLine predicates. In what follows, when we use the same state to evaluate c and $\mathcal{P}(c)$ we mean that these variables have the same value as bitstrings in both programs.
- Predicate equivalence: if P is a predicate and s is a state, then $P(s) \equiv (\mathcal{P}(P))(s)$.
- Semantic equivalence for instructions: if c_i denotes a Jasmin statement that executes a Jasmin processor instruction, then $(\mathcal{P}(c_i), s) \Downarrow_{\mathsf{CL}} s' \implies s' \neq \mathsf{error} \implies (c_i, s) \Downarrow s'$
- We note that both simplified Jasmin programs and CryptoLine programs are static single-assignment, which means that each statement updates the state by creating new variables and assigning values to them. This allows for straightforward solutions when handling function calls. If c_i a Jasmin statement that calls function g, then $\mathcal{P}(c_i) \coloneqq \mathsf{assert}(\mathcal{P}(pre(g)))\mathsf{;assume}(\mathcal{P}(post(g)))$.

This last rule is very important because it allows to not always inline functions during the translation to CryptoLine, and allows for a modular proof where each function is proved only once. The following result allows us to use CryptoLine as a decision oracle for point (2) in Theorem 5.

Theorem 6. *Let c be a simplified safe Jasmin program and P Q be arbitrary predicates. Then, if $\mathcal{P}$ guarantees the properties stated above, we have:*[44]

$$\{\mathcal{P}(P)\}\mathcal{P}(c)\{\mathcal{P}(Q)\}_{\mathsf{CL}} \implies \{P\}c\{Q\}_{\mathsf{assert}}$$

Proof. We note that Jasmin programs do not contain recursive functions. Thus we can do the proof by induction using a lexicographical order on the call graph of c and the size of c. If c is empty it is trivial. If c is $i; c'$ where i is a processor instruction or an assert the result follows trivially by induction. If c is an $\mathsf{assume}(e); c'$, and if e evaluates to $\top$ the result follows by induction; if e evaluated to $\bot$ then the trace exists (because c is safe) and is valid (assert) because it includes $(\mathsf{assume}, \bot)$. If c is a function call followed by c', $\mathcal{P}(c)$ does not really evaluate the function: instead it simply introduces new variables and assumes the post condition. We now must refer to Figure 6.4 and observe that

[44] Note that in the context of Theorem 5 we will be instantiating this theorem with $(c, P, Q) \coloneqq (\mathcal{T}(c'), \mathcal{T}(P'), \mathcal{T}(Q'))$ for some Jasmin source (c', P', Q').

```
fn _ec2(reg u64 g j) -> (reg u64, reg u64)
requires #[prover=smt] { g == j}
ensures #[prover=smt] { result.0 == g + j}
{
#[kind=Assert, prover=smt] assert ( g == j);
  g = g + j;      return (g,g); }

fn _ec2 (reg u64 g_id_130, reg u64 j_id_131) -> (reg u64, reg u64)
requires #[prover=smt] {(g_id_130 ==64u j_id_131)}
ensures #[prover=smt] {(g_id_135 ==64u (g_id_130 +64u j_id_131))}
{
  reg u64 g_id_135;
  #[kind="Assert", prover="smt"]
  assert (g_id_130 ==64u j_id_131);
  (_ , _ , _ , _ , _ ,g_id_135) = #ADD_64(g_id_130, j_id_131);
  return (g_id_135, g_id_135); }
```

Figure 6.5: Annotated Jasmin (top) and simplification (bottom).

the callee Jasmin function assumes the pre-condition (which we already establish to be true), so validity is preserved. Also, since we know that the subroutine is well-annotated for asserts and we are in a state where the pre-condition holds, the trace produced by the subroutine can be appended to the previous trace with the guarantee that validity holds overall. Finally, we use the fact that the subroutine asserts the post-condition on the Jasmin side to derive that the state resulting from the call when the subroutine returns is covered by the proof of CryptoLine correctness of c. This means that this Jasmin state can be used continue the proof by evaluating the two programs in lockstep. ■

Pretty-Printing Instructions

We will explain the pretty-printing using simple examples. Consider the example in Figure 6.5.

The pretty printing of the addition instruction generates the following string:

```
adds NONE_____146@uint1 g_135@uint64 g_130@uint64 j_131@uint64;
```

Note the introduction of the temporary variable to handle the carry. This is the default translation; alternatively, it is possible to annotate the source code with a modifier that pretty-prints the translation as:

```
add g_135@uint64 g_130@uint64 j_131@uint64;
```

However, in this case CryptoLine will reject the program as unsafe in case the carry cannot be proved to be zero. Nevertheless, note that in both cases the final value of the output variable is identical to the value it will take in the Jasmin semantics.

We follow the same strategy to model all Jasmin instructions of interest. Furthermore, we do so for two pretty-printing modes: unsigned (shown in the example above) and signed. The difference between the two is whether translated Jasmin variables (for which the notion of signedness is not defined) are translated to signed or unsigned CryptoLine variables.[45] For many operations, the pretty-printing is actually the same independently of signedness. However, for some instructions, such as arithmetic shifts, one needs to take care to ensure that the final value computed by the CryptoLine translation (assuming the program is declared safe) matches the one computed by Jasmin. As an example, the translation of assigning a constant 65535 to a 16-bit variable a in the signed case is `mov a_144@sint16 (-1)@sint16`.

Pretty-Printing Predicates

The pretty-printing of Jasmin predicates to CryptoLine has several interesting aspects. The first aspect is how we handle the distinction between CryptoLine algebraic and range goals. An attribute can be associated with each annotation to tag it as being intended for pretty-printing as an algebraic or range predicate, which allows $\mathcal{P}$ to perform a few checks. These include confirming that only an appropriate subset of Jasmin expressions can occur in each one of them (expressions computed over the integers for algebraic predicates vs expressions computed over Jasmin words/bit vectors for range predicates). For algebraic predicates we enrich the language of expressions using a few abstract operators that permit capturing the CryptoLine semantics of the annotations. We describe some examples:

- u16i, u32i, etc. are abstract predicates mapping Jasmin words to integers. They represent the integer interpretation of a Jasmin word.
- eqmod_int and eqmod are abstract predicates that represent congruences over the integers and polynomials, respectively.
- mon, mon0 are constructors for monomials, which one can use to construct polynomials.

We note that the Jasmin semantics says nothing about the meaning of these abstract operators, but the pretty-printing to CryptoLine actually assigns to them a semantics. For example u16i has the semantics of interpreting a Jasmin variable as a signed integer when using the signed mode of pretty-printing, and as an unsigned integer otherwise. Similarly, eqmod_int(a, b, c) is printed to the eqmod(a, b, c) CryptoLine operator, which over the integers has the semantics of $a - b = 0 \pmod{c}$.

[45] We could have generalized this part of the translation and considered keeping signedness information for each Jasmin variable, but this is not needed for the examples we are interested in here.

```
op u16i = W16.to_sint. (* for proofs using signed interpretation *)
op u16i = W16.to_uint. (* for proofs using unsigned interpretation *)
op eqmod_int(a b m : int) : bool = (a−b) %% m = 0.

clone export Poly.Poly as IntPoly with type coeff = int.
abbrev (^) = PolyComRing.exp.

op mon(ind deg c : int) = c ** X ^ deg.
op mon0(c : int) = c ** X ^ 0.
op eqmod(a b : poly, c : poly list) =
    ∃ (ws : poly list), size ws = size c ∧ foldr (fun (xmi : poly*poly) (acc : poly) =>
        xmi.`1 * xmi.`2 + acc) poly0 (zip ws c) = (a − b).
```

Figure 6.6: EasyCrypt semantics of CryptoLine predicates.

Implications

We now shortly explain how the results in this section relate to the theoretical framework introduced in the previous sections. Consider some Jasmin function that is pretty-printed to CryptoLine and accepted as correct (and similarly for all functions in the call tree). Then we know that the program is well annotated for assertions, but compared to the previous section we have the novelty that these assertions now have the full semantics assigned by $\mathcal{P}$ and CryptoLine to the abstract operators that occur in the Jasmin code. We can therefore apply Theorem 5 and conclude that the semantics of the above result is reflected back to the Jasmin source and, furthermore, that to transform the CryptoLine proof into a total correctness result, it remains only to prove that the program is well annotated for assumes. (If no assume statements exist, then there is nothing left to prove.)

In the following section we explain how we recover the full semantics of the CryptoLine proof in EasyCrypt and, if necessary, discharge the side-conditions concerning assume statements. For this we rely on a simple EasyCrypt library that brings to EasyCrypt the concrete semantics of the abstract operators fixed by $\mathcal{P}$ and CryptoLine. The fact that these semantics are correct is part of our TCB. However, this assumption is easy to validate by human inspection, since the library is very small. For example, Figure 6.6 shows the relevant parts for the operators above.

6.7 Trace Semantics in EasyCrypt

We have implemented a dedicated extraction feature in the Jasmin compiler tool chain, which takes as input an annotated Jasmin program and generates an EasyCrypt file that captures the semantics of what was proved and left as an unproved assumption by CryptoLine. The extraction should be used when all functions in the call tree of the entry-point annotated Jasmin function have been approved by CryptoLine. Specifically, all the call tree is extracted to

```
proc _ec2 (g:W64.t, j:W64.t) : ((W64.t * trace) = {
  old_g <- g;  old_j <- j;  trace__ec2 <- [];
  (* The precondition is assumed *)
  trace__ec2 <- (trace__ec2 ++ [(Assume, (g = j))]);
  (* The assertion is asserted *)
  trace__ec2 <- (trace__ec2 ++ [(Assert, (g = j))]);
  g <- (g + j);
  (* The postcondition is asserted *)
  trace__ec2 <- (trace__ec2 ++ [(Assert, (g = (old_g + old_j)))]);
  return (g), trace__ec2);
}
```

Figure 6.7: EasyCrypt extraction with traces.

EasyCrypt, and it explicitly computes the trace of annotations as defined in the Jasmin semantics. Figure 6.7 shows the result of extracting the Jasmin function from Figure 6.5. The case of function calls is handled also as expected: all functions add their pre-condition as an assumption and assert the pre-conditions of subroutines. Similarly, each function asserts its own post-condition before returning, and assumes the post-condition of subroutines after they return. Note that the EasyCrypt extraction maintains the calls to subroutines, so the traces computed in EasyCrypt exactly match those described in Figure 6.4. The statement verified by CryptoLine translated to EasyCrypt as follows.

```
lemma _ec2_assert _g _j :
  hoare [M._ec2 : (((_j = j) ∧ (_g = g)) ∧ (_g = _j)) ==> (validk Assert (trace res))].
```

These lemmas are admitted with a comment indicating that they should be proved using CryptoLine. The statement that remains to be verified, corresponding to the assumed conditions in CryptoLine is stated as follows:[46]

```
lemma _ec2_assume_ _g _j :
    hoare [M._ec2 : ((_j = j) ∧ (_g = g)) ==> ((_g = _j) => (validk Assume (trace res)))].
```

This proof needs to be completed in EasyCrypt, and it is typically boilerplate, unless the assumes statements in the proof indeed require deep interactive reasoning in EasyCrypt (e.g. as we will see in the next section this can happen because a function was simply assumed to be correct in the CryptoLine proof introducing an assumption of the post-condition prior to returning). We then generate two lemmas, with synthethised EasyCrypt proofs, that permit deriving the total correctness result stated in Theorem 5. To be able to show that the function contract is holding, we need an additional lemma stating that the postcondition is in the trace :

```
lemma _ec2_valid_post _g _j : hoare [M._ec2 : ((_j = j) ∧ (_g = g)) ==>
    ((valid (trace res)) => (res.`1.`1 = (_g + _j)))].

lemma _ec2_spec _g _j : hoare [M._ec2 :
```

[46] Observe the encoding of the precondition of the function as an implication in the post-condition of the Hoare triple. This clearly equivalent formulation allows us to reduce the boiler-plate EasyCrypt code associated with function calls in CryptoLine proofs.

```
(((_j = j) ∧ (_g = g)) ∧ (_g = _j)) ==> (res.`1.`1 = (_g + _j))].
```

The first lemma establishes that a valid trace implies the post-condition, and the second lemma combines all previous proofs to get the total correctness hoare triple.

6.8 Examples

In this section we illustrate the formal verification methodology for Jasmin programs that combines the traditional EasyCrypt-based approach with the Jazzline approach we introduce in this paper.

6.8.1 X25519 and X-Wing

Figure 6.8 shows a snippet of the EasyCrypt specification of X-Wing: the procedure that expands the X-Wing secret key. This specification, makes calls to the EasyCrypt specifications of three lower-level cryptographic primitives: SHAKE-256, X25519 and ML-KEM.

```
op spec_encode_point (q: zp * zp) : W256.t  =
   let q = q.`1 * (ZModpRing.exp q.`2 (p − 2)) in
      W256.of_int (asint q).

op spec_scalarmult_internal (k: zp) (u: W256.t) : W256.t =
  let r = spec_montgomery_ladder k u in
     spec_encode_point (r.`1).

op spec_scalarmult (k: W256.t) (u: W256.t) : W256.t =
   let k = spec_decode_scalar_25519 k in
   let u = spec_decode_u_coordinate u in
      spec_scalarmult_internal (inzp (to_uint u)) k.

op spec_scalarmult_base (k:W256.t) : W256.t =
   spec_scalarmult (k) (W256.of_int(9%Int)).

proc expandDecapsulationKey(sk : secretkey) : expandedSecretKey  = {
      var expanded, coins1, coins2, coins3, pk_M, sk_M, pk_X_256, pk_X, sk_X, k;

      expanded <− SHAKE256_32_96 sk;

      coins1 <− Array32.init (fun (i: int) => expanded[0 + i]);
      coins2 <− Array32.init (fun (i: int) => expanded[32 + i]);
      coins3 <− Array32.init (fun (i: int) => expanded[64 + i]);

      sk_X <− coins3;
      pk_X_256 <− scalarmult_base (W32u8.pack32 (to_list sk_X));
      pk_X <− Array32.of_list W8.zero (W32u8.to_list pk_X_256);

     (pk_M, sk_M) <@ MLKEM.kg_derand(coins1, coins2);
      k <− (sk_M, sk_X, pk_M, pk_X);
      return k;
   }
```

Figure 6.8: Snippet of the X-Wing EasyCrypt specification.

Figure 6.9 also shows the specification of the X25519 curve operations.

```
op spec_decode_scalar_25519 (k:W256.t) =
  let k = k[0   <- false] in
  let k = k[1   <- false] in
  let k = k[2   <- false] in
  let k = k[255 <- false] in
  let k = k[254 <- true ] in k.

op spec_decode_u_coordinate (u:W256.t) = let u = u[255 <- false] in u.

op spec_add_and_double (qx : zp) (nqs : (zp * zp) * (zp * zp)) =
  let x_1 = qx in
  let (x_2, z_2) = nqs.`1 in
  let (x_3, z_3) = nqs.`2 in
  let a  = x_2 + z_2 in
  let aa = a * a in
  let b  = x_2 + (- z_2) in
  let bb = b*b in
  let e = aa + (- bb) in
  let c = x_3 + z_3 in
  let d = x_3 + (- z_3) in
  let da = d * a in
  let cb = c * b in
  let x_3 = (da + cb)*(da + cb) in
  let z_3 = x_1 * ((da + (- cb))*(da + (- cb))) in
  let x_2 = aa * bb in
  let z_2 = e * (aa + (inzp 121665 * e)) in ((x_2,z_2), (x_3,z_3)).

op spec_montgomery_ladder(init : zp, k : W256.t) =
  let nqs0 = ((Zp.one,Zp.zero),(init,Zp.one)) in
  foldl (fun (nqs : (zp * zp) * (zp * zp)) ctr =>
        if spec_ith_bit k ctr
        then spec_swap_tuple (spec_add_and_double init (spec_swap_tuple(nqs)))
        else spec_add_and_double init nqs) nqs0 (rev (iota_ 0 255)).
```

Figure 6.9: Snippet of the X25519 EasyCrypt specification.

We prove equivalence of the X-Wing implementation in Jasmin to this specification. The proof of equivalence relies directly on the relational Hoare logic provided by EasyCrypt and it is done in multiple hops, i.e., we prove equivalence across a sequence of implementations and use transitivity to derive the result we need: this expresses an equivalence between an extraction of the full X-Wing code to EasyCrypt (a single EasyCrypt module) and the X-Wing EasyCrypt specification. The first transformation proves equivalence of the extracted implementation to another one in which all the code corresponding to SHA-3, X25519 and ML-KEM is removed and is replaced to calls to the EasyCrypt modules over which correctness of SHA-3, X25519 and ML-KEM proofs has already been carried out. This proof is straightforward, as EasyCrypt can detect that the code used by X-Wing is syntactically identical to the ones over which the proofs were carried out. We carry out another hop to lower the level of abstraction of the X25519 specification and replace the functional operators that carry out the X-Wing operations with calls to procedures that implement the same operations imperatively—our specification of X25519 has both variants for convenience, and they are proved to be equivalent to enable

using them interchangeably. Once these steps are performed, the top-level code of the X-Wing implementation is making calls to procedures in the exact same points as the X-Wing specification. This means that the proof of correctness can be performed by simply using EasyCrypt's relational hoare logic to argue that equivalence follows from the fact that the called procedures on both sides are themselves equivalent. For the ML-KEM and SHA-3 implementations these equivalences follow from the proofs carried out in [129, 13] and [17], respectively: our proof just applies the EasyCrypt correctness lemmas that were established in these pre-existing developments.[47] The approach introduced in this paper was instrumental in obtaining proofs for the X25519 implementations with minimal effort, as we describe next.

Proving X25519 in EasyCrypt

The correctness proof for curve X25519 was done simultaneously for two implementations: a reference implementation, and an optimized implementation using the mulx, adox and adcx family of x86-64 operation, that enable the interleaving of add-with-carry operations and multiplications. This allows us to showcase the advantages of using EasyCrypt for the high-level parts of such proofs: we are able essentially reuse the proofs for all the architecture-agnostic parts of the implementation, i.e., the non-leaf functions that are orchestrating the finite-field operations in the correct sequence to compute the elliptic curve operations.

Technically, the proofs of both implementations proceed roughly as we described for the X-Wing proof. We construct an EasyCrypt imperative specification of the functions that compute the curve operations and prove this equivalent to the functional counterpart. This now moves our functional correctness target to a specification of X25519 that has a procedure call-tree that is very close to that adopted by the Jasmin implementations. Indeed, all functions except the leaves are essentially managing calls to lower-level functions, and this structure is the same for both the reference and optimized Jasmin implementations.

The leaf functions perform the same computation in both implementations and therefore satisfy the same contract. However, the way in which the computation is carried out, and therefore the proofs, diverge. At this point, we identify those proofs that are algebraically rich, and for which closely matching CryptoLine proofs already exists. The exceptions are simple functions such as those that fix some bits in the representations of values and/or perform type conversions to/from byte arrays. These latter functions we prove directly in EasyCrypt, and the proofs are straightforward. This highlights another advantage of using EasyCrypt: we have access in the same tool

[47] Some effort was required to update these pre-existing proofs to the more recent version of EasyCrypt and streamline the modular composition of proofs, but these changes were boilerplate refactorings.

and simultaneously to a bit-level semantics and to a high-level integer-based semantics of word operations, so we can choose the most convenient one to carry out a proof.

The remaining leaf functions comprise addition, subtraction, multiplication, squaring and reduction modulo 2^{255}-19. For all operations, the inputs and outputs are represented as 4x64-bit registers, and the contracts are of the general form shown below for the multiplication operation:

```
hoare [M.__mul4_rsr :
  inzpRep4 fs = _f ∧ inzpRep4 g = _g ==> inzpRep4 res = _f * _g ].
```

Here, inzpRep4 is an EasyCrypt operator that takes an array of four 64-bit words, reconstructs the represented multi-precision integer and injects it into the finite field $\mathbb{F}_{2^{255}-19}$. The contract states that the result represents an integer that is congruent to the product of the (integers represented by the) input values, modulo 2^{255}-19.

Using Jazzline to conclude the proof

We conclude the proof by creating an annotated version of each Jasmin function that we want to prove. We show snippets of the multiplication code in Figure 6.10. We can see on the bottom the contract expressing that the function guarantees the desired congruence, expressed here over the integers. On the top side we can see one of the subroutines where we use the strategy of proving using range predicates (i.e., SMT) that a carry flag is zero, and then assume this same property in the algebraic side of the proof.

Throughout, we are using the default translation of instructions to CryptoLine and unsigned interpretation for all variables. CryptoLine accepts the proof, which means that we can extract the result to EasyCrypt and obtain a statement of this result as described in Section 6.7. The top-level EasyCrypt theorem, which we are able to derive with essentially no effort is shown in Figure 6.11.

We note, however that this result applies to an EasyCrypt module that is instrumented with all the operational code required to capture the semantics of asserts and assumes, whereas we need the result to hold over the EasyCrypt extraction of the actual implementation that we are proving correct. However, this final result is trivial to obtain in EasyCrypt by proving that the two versions of the EasyCrypt code (the non-instrumented and the instrumented one) are, in fact, computing the same result. Again, EasyCrypt is able to easily determine that the parts of the code that carry out the computation are syntactically identical.[48]

Lastly, for some of the low-level procedures, such as the modular reduction leaf function, we illustrate another possibility for composable reasoning with

[48]We will see in the next example, however, that for some proofs relying on CryptoLine, it is useful to annotate a Jasmin program that is actually not trivially equivalent to the one for which the proof is intended.

CryptoLine proofs. The proof requires a case analysis, considering for each case that input is in a particular range, such as 0 and $2^{255} - 19$. Hence, we proved the function's correctness multiple times using CryptoLine, with different assert/assume annotations, capturing conditions that follow from each specific case, and that enable an automatic proof. We can then compose these proofs in EasyCrypt and obtain a result that covers the entire range of possible inputs.

```
inline fn __mul4_c1( reg u64[4] h r g, reg u64 f z, reg bool cf of)
  -> reg u64[4],reg u64[4],reg bool,reg bool {
  reg u64 hi lo;
  ( hi, lo )   = #MULX ( f, g[0] );
  of, h[1]     = #ADOX ( h[1], lo, of );
  cf, h[2]     = #ADCX ( h[2], hi, cf );
  ( hi, lo )   = #MULX ( f, g[1] );
  of, h[2]     = #ADOX ( h[2], lo, of );
  cf, h[3]     = #ADCX ( h[3], hi, cf );
  ( hi, lo )   = #MULX ( f, g[2] );
  of, h[3]     = #ADOX ( h[3], lo, of );
  cf, r[0]     = #ADCX ( r[0], hi, cf );
  ( r[1], lo ) = #MULX ( f, g[3] );
  of, r[0]     = #ADOX ( r[0], lo, of);
  cf, r[1]     = #ADCX ( r[1], z, cf);
  of, r[1]     = #ADOX ( r[1], z, of);
  #[kind=Assert, prover=smt] assert (!cf);
  #[kind=Assume, prover=cas] assert (b2i(cf) == 0);
  #[kind=Assert, prover=smt] assert (!of);
  #[kind=Assume, prover=cas] assert (b2i(of) == 0);
  return h, r, cf, of;
}

fn __mul4_rsr(stack u64[4] fs, reg u64[4] g) -> reg u64[4]
ensures #[prover=cas] {
  eqmod (
    \sum (ii ∈ 0:4) (pow(2, 64*ii)*u64i(result.0[ii])),
    \sum (ii ∈ 0:4) (pow(2, 64*ii)*u64i(fs[ii])) *
        \sum (ii ∈ 0:4) (pow(2, 64*ii)*u64i(g[ii])),
    single((pow(2,255)) − 19)
    ) }
{
  g0 = #copy(g);  z = 0;
  f = fs[0];
  h, r, cf, of = __mul4_c0(      f, g0, z, cf, of);
  f = fs[1];
  h, r, cf, of = __mul4_c1(h, r, f, g0, z, cf, of);
  f = fs[2];
  h, r, cf, of = __mul4_c2(h, r, f, g0, z, cf, of);
  f = fs[3];
  h, r, cf, of = __mul4_c3(h, r, f, g0, z, cf, of);
  _38 = 38;
  h = __reduce4(h, r, _38, z, cf, of);
  return h;
}
```

Figure 6.10: Snippet of the Jazzline proof for the X25519 mulx multiplication.

```
lemma __mul4_rsr_spec :
    ∀ _fs _g,
    hoare [M.__mul4_rsr :
    (((_g = g) ∧ (_fs = fs)) ∧ true) ==>
    (eqmod
    (foldr (fun x => (fun (acc: int) => (x + acc))) 0
    (map (fun ii => ((pow 2 (64 * ii)) * (u64i res.`1[ii]))) (iota_ 0 4)))
    ((foldr (fun x => (fun (acc: int) => (x + acc))) 0
    (map (fun ii => ((pow 2 (64 * ii)) * (u64i _fs[ii]))) (iota_ 0 4))) *
    (foldr (fun x => (fun (acc: int) => (x + acc))) 0
    (map (fun ii => ((pow 2 (64 * ii)) * (u64i _g[ii]))) (iota_ 0 4))))
    (single ((pow 2 255) − 19)))].
```

Figure 6.11: EasyCrypt theorem for the X25519 mulx multiplication.

6.8.2 The ML-KEM NTT

Cryptographic constructions based on polynomial rings often rely on the number theoretic transform (NTT) to speed-up multiplications of ring elements. ML-KEM is no exception. They are constructed over the polynomial ring $\mathcal{R}_q := \mathbb{Z}_q[X]/(X^{256}+1)$, where $q = 3329$ is a small prime. Both input and output of NTT can be written as a sequence of 256 coefficients in $\mathbb{Z}_q$. Indeed, the NTT is usually specified as an in-place algorithm over such sequences, which is the way in which implementations are expected to perform the computation. In particular, the FIPS-203 document [167] specifies the algorithm as given in Figure 6.12 (left). A quick look at the right-hand side of Figure 6.12, which includes the Jasmin reference implementation of the NTT routine, shows that the high-level control flow of both routines is very close, and the main difference between specification and implementations lies in the fact that the former is manipulating polynomial coefficients, which are field elements, whereas the implementation is manipulating 16-bit words that represent the corresponding field elements. In the EasyCrypt proofs of ML-KEM implementations carried out in [129, 13] this closeness is explored to construct a direct equivalence proof between two programs in EasyCrypt (ML-KEM specification and model of the Jasmin program) using relational Hoare logic. The non-trivial part of that proof concerns the lower level functions fqmul and poly_reduce, which perform multiplication modulo q in Montgomery representation and coefficient-wise Barrett reduction, respectively.

Algebraic proofs of the NTT in CryptoLine

The above equivalence proof does not capture the mathematical semantics of the computed NTT function: intuitively, it just shows that the two EasyCrypt algorithms (specification and model of implementation) compute the same result. CryptoLine proofs of the ML-KEM NTT are expressed differently: they establish that the output of the NTT operation can be seen as 128 degree 1 polynomials, as given by the formula below, also extracted from FIPS-203.

$$(f \pmod{X^2 - \zeta^{2\mathrm{BitRev}_7(0)+1}}, \ldots, f \pmod{X^2 - \zeta^{2\mathrm{BitRev}_7(127)+1}})$$

Algorithm 9 NTT(f)

Computes the NTT representation $\hat{f}$ of the given polynomial $f \in R_q$.

Input: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the input polynomial
Output: array $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the NTT of the input polynomial

1: $\hat{f} \leftarrow f$ ▷ will compute in place on a copy of input array
2: $i \leftarrow 1$
3: **for** ($len \leftarrow 128$; $len \geq 2$; $len \leftarrow len/2$)
4: **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5: $zeta \leftarrow \zeta^{\text{BitRev}_7(i)} \bmod q$
6: $i \leftarrow i + 1$
7: **for** ($j \leftarrow start$; $j < start + len$; j++)
8: $t \leftarrow zeta \cdot \hat{f}[j + len]$ ▷ steps 8-10 done modulo q
9: $\hat{f}[j + len] \leftarrow \hat{f}[j] - t$
10: $\hat{f}[j] \leftarrow \hat{f}[j] + t$
11: **end for**
12: **end for**
13: **end for**
14: **return** $\hat{f}$

```
fn _poly_ntt(reg ptr u16[MLKEM_N] rp) ->
   reg ptr u16[MLKEM_N] {
  zetasp = jzetas; zetasctr = 0; len = 128;
  while (len >= 2) {
    start = 0;
    while (start < 256) {
      zetasctr += 1; zeta = zetasp[(int)zetasctr];
      j = start; cmp = start; cmp += len;
      while (j < cmp) {
        s = rp[(int)j]; m = s; offset = j; offset += len;
        t = rp[(int)offset]; t = __fqmul(t, zeta); m -= t; t += s;
        rp[(int)offset] = m; rp[(int)j] = t; j += 1;
      } start = j; start += len;
    } len >>= 1; }
  rp = __poly_reduce(rp); return rp; }
```

Figure 6.12: NTT as specified in FIPS203 (top) and Jasmin reference implementation (bottom).

Here, f represents the input polynomial, $\zeta = 17$, and $\text{BitRev}(i)$ is the integer represented by bit-reversing the unsigned 7-bit value that represents $i \in [0, 127]$. This result is, on one hand, semantically richer and, on the other hand, straightforward to obtain fully automatically using CryptoLine, under the assumption that the lower level computations corresponding to the fqmul and poly_reduce routines mentioned above are correct. However, establishing the correctness of these lower level routines is not straightforward to derive automatically, except by brute-forcing the computation using SAT. This could be done for ML-KEM, but this does not scale well to schemes that use larger primes, such as Dilithium/ML-DSA. We use the above scenario as motivation to showcase the advantages of our approach.

Bridging imperative and algebraic NTT specifications

We create in EasyCrypt a general theory that relates imperative specifications of the NTT as given in FIPS-203, and used in the formal specification of ML-KEM in [129, 13], to the CRT-based algebraic specification used in CryptoLine. This result, combined with the algebraic post-condition established automatically by CryptoLine for the Jasmin code, yields an alternative proof of functional correctness for the NTT. Crucially, the proof effort required to establish the relation between the imperative and algebraic specifications can be amortized for other implementations of the ML-KEM NTT and for other polynomial-ring based constructions such as ML-DSA.[49] The listing in Figure 6.13 shows the four EasyCrypt lemmas that permit obtaining the desired result. The ntt_equiv lemma shows that the annotated Jasmin program Poly_ntt.M._poly_ntt that was proved using CryptoLine is equivalent to the Jasmin implementation for which we want the final theorem to hold Poly_ntt_orig.M._poly_ntt. Here the equivalence is not a syntactic identity because we needed to replace while loops with unrollable for loops, but the proof is nevertheless straightforward. The _poly_ntt_spec_p lemma is the total correctness version of the CryptoLine result, which we obtain simply by proving termination; again, a straightforward proof. Note that, from CryptoLine we not only get the algebraic specification, but also a range contract that sets the admissible bounds of inputs and the enforced bounds at the outputs. Finally, the main lemma ntt_correct is obtained as a consequence of the above lemmas, and the fact that the ntt operator that captures the semantics of imperative specification is equivalent to the algebraic specification by the result in lemma ntt_correct_core.

Proving parts of the call-tree in EasyCrypt

We show in Figure 6.14 how, in EasyCrypt, we can complete a CryptoLine proof of the NTT function in which the correctness of field multiplication and Barrett reduction subroutines was assumed. Concretely, lemma trans_reduce expresses a relational contract between function M.__barrett_reduce and function Poly_ntt_orig.M.__barrett_reduce. The latter function has been proved in EasyCrypt to satisfy the required contract—the name of this lemma is Fq.barrett_reduce_corr_h. The former function includes an annotated assume statement of the post-condition immediately before the return statement, which guarantees a trivial proof in CryptoLine.

The relational contract establishes a bridge between the known result for the returned value of Poly_ntt_orig.M.__barrett_reduce and the unproved assumption in CryptoLine. Here, we leverage the fact that the semantics of CryptoLine assert and assume statements are captured by global variables in EasyCrypt: this allows us to refer to predicates that are evaluated in the middle of the

[49] For example, we reuse this result for a CryptoLine-based proof of the AVX2 implementation of the ML-KEM NTT, as we will describe later in the paper.

```
lemma ntt_equiv :
  equiv [ Poly_ntt_orig.M._poly_ntt ~ Poly_ntt.M._poly_ntt : ={arg} ==> res{1} = res{2}.`1 ].

lemma _poly_ntt_spec_p _rp:
    phoare [Poly_ntt.M._poly_ntt : _rp = arg ∧
    foldr (fun (x acc : bool) => x ∧ acc) true (map (fun (ii : int) =>
          (W16.of_int ((−2) * 3329) \sle arg[ii]) ∧
               (arg[ii] \slt W16.of_int (2 * 3329))) (iota_ 0 256)) ==>
    foldr (fun (x acc : bool) => x ∧ acc) true
      (map (fun (ii : int) => (W16.zero \sle res.`1[ii]) ∧
                (res.`1[ii] \slt W16.of_int (2*3329))) (iota_ 0 256)) ∧
    foldr (fun (x acc : bool) => x ∧ acc) true (map (fun (pp : int) =>
      eqmod (foldr (fun (x acc : poly) => x + acc) (mon0 0)
       (map (fun (ii : int) => mon 0 ii (u16i _rp[ii])) (iota_ 0 256)))
        (mon0 (u16i res.`1[pp * 2]) + mon 0 1 (u16i res.`1[pp * 2 + 1])) (pair (mon0 3329)
             (mon 0 2 1 − mon0 (u16i Poly_ntt.zetas_power[pp]))))) (iota_ 0 128))] = 1%r

lemma ntt_correct_core (_r : Zq.coeff Array256.t) (_rp p: W16.t Array256.t) :
    _r = lift_array256 _rp =>
    foldr (fun (x acc : bool) => x ∧ acc) true
     (map (fun (pp : int) => eqmod (foldr (fun (x acc : poly) => x + acc) (mon0 0)
              (map (fun (ii : int) => mon 0 ii (u16i _rp[ii])) (iota_ 0 256)))
            (mon0 (u16i p[pp * 2]) + mon 0 1 (u16i p[pp * 2 + 1])) (pair (mon0 3329)
            (mon 0 2 1 − mon0 (u16i zetas_power[pp]))))) (iota_ 0 128)) => ntt _r = lift_array256 p.

lemma ntt_correct _r :
  phoare[M._poly_ntt :
      _r = lift_array256 rp ∧ signed_bound_cxq rp 0 256 2 ==> ntt _r = lift_array256 res ∧
         ∀ k, 0≤k<256 => bpos16 res[k] (2*q)] = 1%r.
eclim rp => _rp; conseq ntt_equiv (_poly_ntt_spec_p _rp).
```

Figure 6.13: EasyCrypt lemmas for NTT functional correctnes.

annotated program (in this case just before the return statement) and relate them to the outputs of the program we have proved in EasyCrypt.

Concretely, the trans_reduce lemma says that, if both functions start from the same input, then both will produce the same result but, moreover, the flag that encodes the assumed predicate res1.`2.`3 is equivalent to the post-condition over the result. This permits deriving a proof for the assumed predicate in lemma __barrett_reduce_assume simply as a consequence of the above relation and the fact that the result of the Poly_ntt_orig.M.__barrett_reduce function has already been shown to be correct.

```
equiv trans_reduce _a : M.__barrett_reduce ~ Poly_ntt_orig.M.__barrett_reduce :
  ={arg} ∧ to_sint arg{1} = _a
   ==> res{1}.`1 = res{2} ∧
      (validk Assume res{1}.`2 ⇔ (((((W16.of_int 0) \sle res{1}.`1) ∧
         (res{1}.`1 \slt (W16.of_int (2*3329))))) ∧ (eqmod_int (u16i res{1}.`1) (_a) 3329))).

lemma __barrett_reduce_assume :
    hoare [M.__barrett_reduce : true ==> (assume_proof_ res)].
proof.
eclim arg => _a.
conseq (trans_reduce (to_sint _a)) (Fq.barrett_reduce_corr_h (to_sint _a)) .
```

Figure 6.14: EasyCrypt proof of leaf functions.

Extension to vectorized instructions.

We now describe further extensions to our generation of CryptoLine proofs from annotated Jasmin programs that allow us to take advantage of the vectorized extension to CryptoLine. The motivating example is the AVX2 implementation of the NTT. The reference implementation of the NTT function that we discussed in Section 6.8 does not fully show the advantages of the automation offered by CryptoLine wrt a fully interactive proof in EasyCrypt. This is because, as we mentioned above, the specification and implementation are close enough to allow an interactive proof with reasonable effort. This is *not* the case for the AVX2 implementation, where there is no discernable relation between the control flow of the specification and the implementation. For this reason, being able to establish automatically that the implementation computes the algebraic specification indeed provides a huge benefit wrt to the overall proof effort required for an interactive proof. However, the use of vectorized instructions in this implementation requires that we enrich our interaction with CryptoLine to permit reasoning about vectorized instructions. We first briefly explain the support CryptoLine provides at this level.

Vectorized instructions in CryptoLine

CryptoLine does not have support for architecture-specific vectorized extensions such as AVX2. Instead, CryptoLine offers a generic mechanism to describe programs that rely on vectorized operations. This mechanism, in very short terms, works as follows. Program variables can be typed as vectors of scalar variables, e.g., one can initialize a variable $\%v$, whereby the percentage symbol denotes a vector, with a list of scalar variables $[v_1, \ldots, v_n]$, where all variables in the vector must be of the same CryptoLine type. Conversely, a vector $\%v$ can be assigned to a list of scalar variables of the appropriate type, or it can be *cast* into a list of variables of different types, conditionally on the natural size restrictions being satisfied. Furthermore, CryptoLine offers the natural lift of scalar operations to vectors by interpreting their application pointwise. The vectorized instruction is signalled with a % sign. Operations that break the trivial parallel computation pattern can be emulated by *unserializing* a vector into a set of scalar variables and performing computations over the scalar variables using the standard CryptoLine approach. For example, a permutation can be modelled by assigning the vector to scalar variables and initializing a new vector with the permuted scalars.

The problem

A simplified Jasmin program with vector instructions does not necessarily have a natural pretty-printed CryptoLine counterpart. The problem is that a CryptoLine vector variable has a fixed type for its contents, whereas a long register in Jasmin can be used as an input to instructions that will handle its

contents with different vector types. One possible solution to this problem would be to take the (certified) simplification of the Jasmin program one step further and convert it into a scalar program over which we could apply the approach we described in the previous examples. This is a route we leave to explore in future work. In this paper, we give a simple solution that requires placing a slightly more trust in the pretty printing routine.

Our solution

We enrich our translation to CryptoLine with knowledge on how how each AVX2 instruction interprets the vector register as a list of scalar values of specific size (e.g VPADD_16u16 treats a 256-bit register as a vector of 16x16-bit values). This means that we can guarantee the generation of a well-typed CryptoLine program as follows. We maintain the invariant that all variables of a long word type (i.e. greater than the machine word size) are initialized in CryptoLine as vector variables on their first assignment, i.e. their type is defined by the operation that first assigns to them.[50] Then, each AVX2 instruction is pretty printed as follows: for each vector input argument, it uses the cast operation in CryptoLine to create a list of scalar variables representing the input. The types of these scalar variables are determined by the AVX2 instruction that will be computed. These scalar variables are then packed into a new (fresh variable) vector input that can be passed to the vectorized CryptoLine instruction. In this way, even if adjacent AVX2 instructions are processing long registers with different vector granularity, the output CryptoLine program will be valid. The disadvantage of this approach is that the resulting program may certainly include unnecessary cast operations: this will happen whenever two adjacent vector instructions are operating over the same sub-word size (e.g. VPADD_16u16 followed by VPSUB_16u16). In order to reduce the complexity of the model, we introduced a non-verified simplification in our pretty-printing routine that identifies and removes redundant casts. This simple heuristic works well for the AVX2 implementation of the NTT, which is very well behaved wrt its use of SIMD operations—all vectorized instructions work over 256-bit words seen as 16x16-bit sub-words, so essentially all casts are removed. A final extension to the pretty-printing mechanism was necessary to allow writing Jasmin annotations and contracts that refer to sub-words of long inputs and outputs. To this end, we allow the Jazzline user to annotate the types of long inputs and outputs to functions with their corresponding vector type and introduce abstract predicates that permit referring to subwords in the syntax of both range and algebraic predicates. This meta-information informs the pretty-printing mechanism of how long inputs and outputs to a Jasmin function should be partitioned into scalar variables: the resulting CryptoLine program takes scalar variables as inputs and outputs, so its contract is written in a purely scalar form. The first/last instructions in the pretty-printed program perform

[50]The exception here are input and output variables, which we explain later.

the necessary packing/unpacking to guarantee consistency. Similarly, assert and assume annotations are expressed over scalar variables resulting from unpacking instructions.

The proof.

To demonstrate the feasiblity of this extension we targeted the NTT implementation of ML-KEM-768. The approach here largely follows the one described for the reference implementation of the same function, the main difference being the way the pre- and post-conditions are specified. As stated previously, the use of input polynomials as arrays of vector registers instead of arrays of scalar values, requires us to use an unpacking predicate (in this case `u256_as_16u16`) and nested big operators in order to index each coefficient individually. The use of EasyCrypt in the proof of the AVX2 function is very close to what we described for the scalar version of the algorithm. Matching the algebraic contract resulting from the CryptoLine proof to the intended EasyCrypt contract follows by applying the same lemma relating the algebraic definition of the NTT to its imperative specification. We also factored out the proof of the Barrett reduction step into a proof goal that was assumed in CryptoLine and discharged interactively in EasyCrypt. Moreover, we did the same for some subroutines that are performing sequences of permutations in order to reshuffle intermediate values. The take-away message from this example is that our approach is conceptually general enough to allow reusing CryptoLine proofs relying on vectorized instructions for similar Jasmin programs. However, we have also learned that extending the framework to support the wide variety of SIMD instructions included in various architectures will give rise to a complex pretty-printer. As mentioned above, to avoid that, we plan to explore certified compilation steps that generically convert vectorized programs into equivalent scalar ones.

6.9 Conclusion

In this paper we show that it is possible to integrate two formal verification frameworks, in a sound way, to enable proofs that explore the advantages of both. This permits amortizing proof effort, and allows for collaborative work in proof development using different formal verification approaches. As illustration of this fact we present a functionally correct high-speed implementation of X-Wing in Jasmin, where we reused proofs initially done using CryptoLine. In future work we plan to extend the Jasmin compiler to cover a wider range of programs in simplification, and to explore the possibility of automatically transform vectorized instructions into scalar ones in a certified way.

7
Outlook

The results presented in this thesis demonstrate that formal methods can be applied to deployment-ready implementations of post-quantum key-establishment mechanisms, as well as other cryptographic primitives. In particular, the work shows that optimized implementations can be combined with machine-checked security proofs and implementation-correctness proofs to obtain strong guarantees about cryptographic code intended for real-world deployment.

Despite this, and as shown throughout this thesis, the guarantees provided by formal methods remain bounded by the scope of the formal models being used, the properties that are proved, the trusted computing base, and the complexity of the tools involved. These limitations raise questions about how verification results can be composed across abstraction layers and across different tools to further close the gap between formal models, verified implementations, and deployed cryptographic systems.

A useful lens through which to analyze formal verification is to conceptualize its application in two dimensions. The first is a *vertical dimension*, which pertains to the three pillars discussed in Section 2.3. This dimension involves the application of formal methods at different stages of the development process for a single protocol or primitive, with the purpose of attaining different guarantees. The second is a *horizontal dimension*, which pertains to the interaction between different formal verification methods and the systems and components they verify.

With respect to the latter, cryptographic primitives such as KEMs and NIKEs are seldom used in isolation. Rather, they are components of larger systems, including cryptographic protocols, whose security is predicated on the assumption that the underlying blocks and primitives are secure and correct. Combining the results of tools such as Tamarin [188, 160] and ProVerif [58], which focus on the verification of protocols instead of low level primitives, with others that focus on the low level primitives, could enable a "full stack" verification framework. Efforts such as the one in [53] already represent a step in that direction by using the hax framework [56] to verify Bert13, a Rust implementation of TLS1.3, achieving guarantees not only of the individual

primitives but also of the protocol as a whole without needing to assume that the building blocks are secure, correct, or safe.

Another line of work in the horizontal dimension involves combining approaches, such as those in [12] and [16], to verify different parts of the same system. This approach, analogous to the philosophy of the hax framework, enables implementers to select the most suitable technique for a specific verification task, depending on the target subcomponent.

7.1 Formosa toolchain

When considering the vertical dimension, it is also relevant to analyze the toolchain being used, determining which parts of it are verified and which ones are part of the trusted computing base. In the case of the Formosa toolchain, as discussed in Section 2.3.2, the latter includes the model of the underlying architecture (e.g., the x86 instruction semantics in EasyCrypt and Jasmin), the extraction from Jasmin to EasyCrypt, and the extraction from Jasmin to CryptoLine, which are not verified. The identification of techniques to minimize this trusted computing base, namely through additional verification initiatives, serves to address the gap between the models, formal proofs, and the code that is actually executed. Specifically, formalizing the extraction to EasyCrypt as well as the one to CryptoLine, to ensure semantic equivalence, would close gaps and reinforce trust in formally verified implementations using the Formosa toolchain.

Finally, a limitation of the Formosa toolchain, particularly the Jasmin framework [15], is the lack of portability of its implementations and, consequently, of the proofs that reason about these implementations. This limitation stems mainly from the fact that Jasmin implementations are generally targeted at a specific architecture. Consequently, when attempting to port verified implementations to a different architecture, significant portions of correctness proofs must be rewritten. Once more, endeavors such as [12, 16] serve to reduce the effort required to rewrite these proofs for new parameter sets and architectures. Nevertheless, efforts to add support for architecture-agnostic Jasmin implementations that are portable and can be verified once for multiple architectures represent a step towards facilitating the deployment of formal methods in the "real world".

Overall, these directions point toward a broader goal: making high-assurance cryptography practical not only for isolated primitives, but for complete cryptographic systems deployed in real-world environments.

Bibliography

[1] Carlos Aguilar Melchor et al. *Hamming Quasi-Cyclic (HQC)*. https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf. 2025 (cit. on p. 2).

[2] Miklós Ajtai. "Generating Hard Instances of Lattice Problems (Extended Abstract)". In: *28th Annual ACM Symposium on Theory of Computing*. Philadephia, PA, USA: ACM Press, May 1996, pp. 99–108. DOI: 10.1145/237814.237838 (cit. on p. 6).

[3] Gorjan Alagic et al. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. NISTIR 8413. https://csrc.nist.gov/publications/detail/nistir/8413/final. 2022 (cit. on pp. 30, 65).

[4] Martin Albrecht and Léo Ducas. *Lattice Attacks on NTRU and LWE: A History of Refinements*. Cryptology ePrint Archive, Report 2021/799. 2021. URL: https://eprint.iacr.org/2021/799 (cit. on p. 68).

[5] Martin R. Albrecht. "On Dual Lattice Attacks Against Small-Secret LWE and Parameter Choices in HElib and SEAL". In: *Advances in Cryptology – EUROCRYPT 2017, Part II*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10211. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 103–129. DOI: 10.1007/978-3-319-56614-6_4 (cit. on p. 68).

[6] Martin R. Albrecht, Rachel Player and Sam Scott. *On The Concrete Hardness Of Learning With Errors*. Cryptology ePrint Archive, Report 2015/046. https://eprint.iacr.org/2015/046. 2015 (cit. on p. 120).

[7] Martin R. Albrecht et al. *Classic McEliece*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions. National Institute of Standards and Technology, 2022 (cit. on pp. 102, 122).

[8] Martin R. Albrecht et al. "Estimate All the LWE, NTRU Schemes!" In: *SCN 18: 11th International Conference on Security in Communication Networks*. Ed. by Dario Catalano and Roberto De Prisco. Vol. 11035. Lecture Notes in Computer Science. Amalfi, Italy, Sept. 2018, pp. 351–367. DOI: 10.1007/978-3-319-98113-0_19 (cit. on p. 120).

[9] Martin R. Albrecht et al. *Estimate all the LWE, NTRU schemes!* Cryptology ePrint Archive, Report 2018/331. https://eprint.iacr.org/2018/331. 2018 (cit. on p. 121).

[10] Martin R. Albrecht et al. "The General Sieve Kernel and New Records in Lattice Reduction". In: *Advances in Cryptology – EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Cham, Switzerland, May 2019, pp. 717–746. DOI: 10.1007/978-3-030-17656-3_25 (cit. on p. 68).

[11] Erdem Alkim et al. "Post-quantum Key Exchange - A New Hope". In: *USENIX Security 2016: 25th USENIX Security Symposium*. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 327–343 (cit. on pp. 100, 110, 123, 124).

[12] José Bacelar Almeida et al. "Faster Verification of Faster Implementations: Combining Deductive and Circuit-Based Reasoning in EasyCrypt". In: *2025 IEEE Symposium on Security and Privacy*. Ed. by Marina Blanton, William Enck and Cristina Nita-Rotaru. San Francisco, CA, USA: IEEE Computer Society Press, May 2025, pp. 3820–3838. DOI: 10.1109/SP61157.2025.00214 (cit. on p. 161).

[13] José Bacelar Almeida et al. "Formally Verifying Kyber - Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt". In: *Advances in Cryptology – CRYPTO 2024, Part II*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14921. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2024, pp. 384–421. DOI: 10.1007/978-3-031-68379-4_12 (cit. on pp. 3, 64, 128, 132, 150, 153, 155, 207).

[14] José Bacelar Almeida et al. *Formally verifying Kyber Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt*. Cryptology ePrint Archive, Report 2024/843. 2024. URL: https://eprint.iacr.org/2024/843 (cit. on pp. 71, 82, 85, 86, 89).

[15] José Bacelar Almeida et al. "Jasmin: High-Assurance and High-Speed Cryptography". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani Thuraisingham et al. https://doi.org/10.1145/3133956.3134078. ACM, 2017, pp. 1807–1823. DOI: 10.1145/3133956.3134078. URL: https://doi.org/10.1145/3133956.3134078 (cit. on pp. 19, 33, 65, 102, 123, 161).

[16] José Bacelar Almeida et al. "Jazzline: Composable CryptoLine Functional Correctness Proofs for Jasmin Programs". In: *ACM CCS 2025: 32nd Conference on Computer and Communications Security*. ACM Press, Nov. 2025, pp. 1409–1423. DOI: 10.1145/3719027.3744814 (cit. on pp. 4, 127, 161, 208).

[17] José Bacelar Almeida et al. "Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3". In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Ed. by Lorenzo Cavallaro et al. London, UK: ACM Press, Nov. 2019, pp. 1607–1622. DOI: 10.1145/3319535.3363211 (cit. on pp. 34, 42, 44, 68, 128, 132, 150, 191).

[18] José Bacelar Almeida et al. "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations". In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 965–982. DOI: 10.1109/SP40000.2020.00028 (cit. on pp. 33, 65, 102, 123).

[19] José Bacelar Almeida et al. "Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC". In: *Fast Software Encryption – FSE 2016*. Ed. by Thomas Peyrin. Vol. 9783. Lecture Notes in Computer Science. Bochum, Germany: Springer Berlin Heidelberg, Germany, Mar. 2016, pp. 163–184. DOI: 10.1007/978-3-662-52993-5_9 (cit. on pp. 68, 96).

[20] Andris Ambainis, Mike Hamburg and Dominique Unruh. "Quantum Security Proofs Using Semi-classical Oracles". In: *Advances in Cryptology – CRYPTO 2019, Part II*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11693. Lecture Notes in Computer Science. Santa Barbara, CA, USA, Aug. 2019, pp. 269–295. DOI: 10.1007/978-3-030-26951-7_10 (cit. on p. 109).

[21] Basavesh Ammanaghatta Shivakumar et al. "Typing High-Speed Cryptography against Spectre v1". In: *2023 IEEE Symposium on Security and Privacy (SP)*. (to appear) https://eprint.iacr.org/2022/1270. IEEE Computer Society, 2023 (cit. on pp. 2, 34, 62).

[22] Yawning Angel et al. "Post Quantum Noise". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin et al. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 97–109. DOI: 10.1145/3548606.3560577 (cit. on p. 101).

[23] Andrew W. Appel. "Verification of a Cryptographic Primitive: SHA-256". In: *ACM Trans. Program. Lang. Syst.* 37.2 (2015), 7:1–7:31. DOI: 10.1145/2701415. URL: https://doi.org/10.1145/2701415 (cit. on p. 131).

[24] Jean-Philippe Aumasson et al. *SPHINCS+: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project [170]. https://sphincs.org/. 2020 (cit. on p. 2).

[25] Jean-Philippe Aumasson et al. *SPHINCS⁺ – Submission to the NIST post-quantum project, v.3.1*. Round-3 submission to the NIST PQC standardization project. https://sphincs.org/data/sphincs+-r3.1-specification.pdf. 2022 (cit. on p. 30).

[26] Roberto Avanzi et al. *CRYSTALS–Kyber: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project [170]. https://pq-crystals.org/kyber. 2017 (cit. on p. 2).

[27] Roberto Avanzi et al. *CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation (version 3.02)*. Round-3 submission to the NIST PQC standardization project. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf. 2021 (cit. on pp. 31, 32, 35, 36, 43, 49, 65).

[28] Reza Azarderakhsh, David Jao and Christopher Leonardi. "Post-Quantum Static-Static Key Agreement Using Multiple Protocol Instances". In: *SAC 2017: 24th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Carlisle Adams and Jan Camenisch. Vol. 10719. Lecture Notes in Computer Science. Ottawa, ON, Canada, Aug. 2017, pp. 45–63. DOI: 10.1007/978-3-319-72565-9_3 (cit. on p. 103).

[29] Gustavo Banegas et al. "CTIDH: faster constant-time CSIDH". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (2021). https://tches.iacr.org/index.php/TCHES/article/view/9069, pp. 351–387. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.351-387 (cit. on pp. 103, 122, 125, 126).

[30] Manuel Barbosa and Andreas Hülsing. *The security of Kyber's FO-transform*. Cryptology ePrint Archive, Report 2023/755. https://eprint.iacr.org/2023/755. 2023 (cit. on p. 73).

[31] Manuel Barbosa et al. "A Tight Security Proof for SPHINCS⁺, Formally Verified". In: *Advances in Cryptology – ASIACRYPT 2024, Part IV*. Ed. by Kai-Min Chung and Yu Sasaki. Vol. 15487. Lecture Notes in Computer Science. Kolkata, India: Springer, Singapore, Singapore, Dec. 2024, pp. 35–67. DOI: 10.1007/978-981-96-0894-2_2 (cit. on p. 17).

[32] Manuel Barbosa et al. "EasyPQC: Verifying Post-Quantum Cryptography". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 2021, pp. 2564–2586. DOI: 10.1145/3460120.3484567 (cit. on pp. 35, 68).

[33] Manuel Barbosa et al. “Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium”. In: *Advances in Cryptology – CRYPTO 2023, Part V*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14085. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2023, pp. 358–389. DOI: 10.1007/978-3-031-38554-4_12 (cit. on p. 68).

[34] Manuel Barbosa et al. “Machine-Checked Security for XMSS as in RFC 8391 and SPHINCS$^+$”. In: *Advances in Cryptology – CRYPTO 2023, Part V*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14085. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2023, pp. 421–454. DOI: 10.1007/978-3-031-38554-4_14 (cit. on pp. 17, 68).

[35] Manuel Barbosa et al. “SoK: Computer-Aided Cryptography”. In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008 (cit. on pp. 2, 16, 18, 31, 35, 65, 68, 128, 131).

[36] Manuel Barbosa et al. “X-Wing”. In: *IACR Communications in Cryptology (CiC)* 1.1 (2024), p. 21. DOI: 10.62056/a3qj89n4e (cit. on p. 132).

[37] Gilles Barthe et al. “Computer-Aided Security Proofs for the Working Cryptographer”. In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2011, pp. 71–90. DOI: 10.1007/978-3-642-22792-9_5 (cit. on pp. 17, 19, 32, 65, 132).

[38] Gilles Barthe et al. “EasyCrypt: A Tutorial”. In: *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier Lopez and Fabio Martinelli. . Springer, 2014, pp. 146–166 (cit. on p. 71).

[39] Gilles Barthe et al. “Symbolic Proofs for Lattice-Based Cryptography”. In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie et al. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 538–555. DOI: 10.1145/3243734.3243825 (cit. on p. 68).

[40] Anja Becker et al. “New directions in nearest neighbor searching with applications to lattice sieving”. In: *SODA ’16: Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2016, pp. 10–24 (cit. on p. 68).

[41] Michiel Van Beirendonck et al. “A Side-Channel-Resistant Implementation of SABER”. In: *J. Emerg. Technol. Comput. Syst.* 17.2 (Apr. 2021). https://eprint.iacr.org/2020/733 (cit. on p. 30).

[42] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *ACM CCS 93: 1st Conference on Computer and Communications Security*. Ed. by Dorothy E. Denning et al. Fairfax, Virginia, USA: ACM Press, Nov. 1993, pp. 62–73. DOI: 10.1145/168588.168596 (cit. on pp. 108, 194).

[43] Mihir Bellare and Phillip Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *Advances in Cryptology – EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. Lecture Notes in Computer Science. St. Petersburg, Russia, May 2006, pp. 409–426. DOI: 10.1007/11761679_25 (cit. on p. 107).

[44] Lennart Beringer et al. "Verified Correctness and Security of OpenSSL HMAC". In: *USENIX Security 2015: 24th USENIX Security Symposium*. Ed. by Jaeyeon Jung and Thorsten Holz. Washington, DC, USA: USENIX Association, Aug. 2015, pp. 207–221 (cit. on p. 68).

[45] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Moti Yung et al. Vol. 3958. Lecture Notes in Computer Science. New York, NY, USA, Apr. 2006, pp. 207–228. DOI: 10.1007/11745853_14 (cit. on pp. 122, 125, 126).

[46] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. http://bench.cr.yp.to/ (cit. on pp. 60, 126).

[47] Daniel J. Bernstein and Edoardo Persichetti. *Towards KEM Unification*. Cryptology ePrint Archive, Report 2018/526. 2018. URL: https://eprint.iacr.org/2018/526 (cit. on p. 83).

[48] Daniel J. Bernstein et al. "OpenSSLNTRU: Faster post-quantum TLS key exchange". In: *USENIX Security 2022: 31st USENIX Security Symposium*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 845–862 (cit. on p. 101).

[49] Daniel J. Bernstein et al. "Quantum Circuits for the CSIDH: Optimizing Quantum Evaluation of Isogenies". In: *Advances in Cryptology – EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Darmstadt, Germany, May 2019, pp. 409–441. DOI: 10.1007/978-3-030-17656-3_15 (cit. on p. 103).

[50] Daniel J. Bernstein et al. "The SPHINCS$^+$ Signature Framework". In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Ed. by Lorenzo Cavallaro et al. London, UK: ACM Press, Nov. 2019, pp. 2129–2146. DOI: 10.1145/3319535.3363229 (cit. on pp. 17, 30).

[51] Benjamin Beurdouche. *Verified cryptography for Firefox 57*. Post on the Mozilla Security Blog. https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/ (accessed: 2025-10-10). 2017 (cit. on pp. 2, 31).

[52] Karthikeyan Bhargavan. *Helping Secure Signal's Post-Quantum Transition*. https://cryspen.com/post/signal-spqr-verification/ (accessed 2025-10-02). 2025 (cit. on p. 2).

[53] Karthikeyan Bhargavan et al. "Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust". In: *ACM CCS 2025: 32nd Conference on Computer and Communications Security*. ACM Press, Nov. 2025, pp. 2729–2743. DOI: 10.1145/3719027.3765213 (cit. on p. 160).

[54] Karthikeyan Bhargavan et al. "Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging". In: *USENIX Security 2024: 33rd USENIX Security Symposium*. Ed. by Davide Balzarotti and Wenyuan Xu. Philadelphia, PA, USA: USENIX Association, Aug. 2024 (cit. on p. 2).

[55] Karthikeyan Bhargavan et al. *hax: Verifying Security-Critical Rust Software using Multiple Provers*. Cryptology ePrint Archive, Paper 2025/142. 2025. URL: https://eprint.iacr.org/2025/142 (cit. on p. 17).

[56] Karthikeyan Bhargavan et al. *hax: Verifying Security-Critical Rust Software using Multiple Provers*. Cryptology ePrint Archive, Report 2025/142. 2025. URL: https://eprint.iacr.org/2025/142 (cit. on p. 160).

[57] Nina Bindel et al. "Tighter Proofs of CCA Security in the Quantum Random Oracle Model". In: *TCC 2019: 17th Theory of Cryptography Conference, Part II*. Ed. by Dennis Hofheinz and Alon Rosen. Vol. 11892. Lecture Notes in Computer Science. Nuremberg, Germany: Springer, Cham, Switzerland, Dec. 2019, pp. 61–90. DOI: 10.1007/978-3-030-36033-7_3 (cit. on p. 68).

[58] Bruno Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif". In: *Found. Trends Priv. Secur.* 1.1–2 (Oct. 2016), pp. 1–135. ISSN: 2474-1558. DOI: 10.1561/3300000004. URL: https://doi.org/10.1561/3300000004 (cit. on pp. 17, 160).

[59] J. Blanchette and A. Mahboubi, eds. *Handbook of Proof Assistants*. To appear. Springer-Verlag, 2025 (cit. on p. 68).

[60] Olivier Blazy and Céline Chevalier. "Non-Interactive Key Exchange from Identity-Based Encryption". In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ARES 2018. Hamburg, Germany: Association for Computing Machinery, 2018. ISBN: 9781450364485. DOI: 10.1145/3230833.3230864. URL: https://doi.org/10.1145/3230833.3230864 (cit. on p. 103).

[61] Manuel Blum, Paul Feldman and Silvio Micali. "Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract)". In: *20th Annual ACM Symposium on Theory of Computing*. Chicago, IL, USA: ACM Press, May 1988, pp. 103–112. DOI: 10.1145/62212.62222 (cit. on p. 194).

[62] Dan Boneh and Mark Zhandry. "Multiparty Key Exchange, Efficient Traitor Tracing, and More from Indistinguishability Obfuscation". In: *Advances in Cryptology – CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. Lecture Notes in Computer Science. Santa Barbara, CA, USA, Aug. 2014, pp. 480–499. DOI: 10.1007/978-3-662-44371-2_27 (cit. on p. 103).

[63] Dan Boneh et al. "Random Oracles in a Quantum World". In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. Lecture Notes in Computer Science. Seoul, South Korea: Springer Berlin Heidelberg, Germany, Dec. 2011, pp. 41–69. DOI: 10.1007/978-3-642-25385-0_3 (cit. on pp. 68, 69, 108).

[64] Xavier Bonnetain and André Schrottenloher. "Quantum Security Analysis of CSIDH". In: *Advances in Cryptology – EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Zagreb, Croatia, May 2020, pp. 493–522. DOI: 10.1007/978-3-030-45724-2_17 (cit. on p. 103).

[65] Joppe W. Bos et al. "CRYSTALS – Kyber: A CCA-Secure Module-Lattice-Based KEM". In: *IEEE European Symposium on Security and Privacy – EuroS&P 2018*. https://eprint.iacr.org/2017/634. IEEE, 2018, pp. 353–367 (cit. on pp. 31, 65, 66, 68, 75–77, 79, 81).

[66] Joppe W. Bos et al. "Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE". In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl et al. Vienna, Austria: ACM Press, Oct. 2016, pp. 1006–1018. DOI: 10.1145/2976749.2978425 (cit. on p. 100).

[67] Joppe W. Bos et al. "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem". In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 2015, pp. 553–570. DOI: 10.1109/SP.2015.40 (cit. on pp. 101, 110).

[68] Colin Boyd et al. “Efficient One-Round Key Exchange in the Standard Model”. In: *ACISP 08: 13th Australasian Conference on Information Security and Privacy*. Ed. by Yi Mu, Willy Susilo and Jennifer Seberry. Vol. 5107. Lecture Notes in Computer Science. Wollongong, Australia, July 2008, pp. 69–83 (cit. on p. 100).

[69] Zvika Brakerski et al. “Classical hardness of learning with errors”. In: *45th Annual ACM Symposium on Theory of Computing*. Ed. by Dan Boneh, Tim Roughgarden and Joan Feigenbaum. Palo Alto, CA, USA: ACM Press, June 2013, pp. 575–584. DOI: 10.1145/2488608.2488680 (cit. on p. 109).

[70] Jacqueline Brendel et al. “Post-quantum Asynchronous Deniable Key Exchange and the Signal Handshake”. In: *PKC 2022: 25th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Goichiro Hanaoka, Junji Shikata and Yohei Watanabe. Vol. 13178. Lecture Notes in Computer Science. Virtual Event, Mar. 2022, pp. 3–34. DOI: 10.1007/978-3-030-97131-1_1 (cit. on p. 102).

[71] Jacqueline Brendel et al. “Towards Post-Quantum Security for Signal’s X3DH Handshake”. In: *SAC 2020: 27th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Orr Dunkelman, Michael J. Jacobson Jr. and Colin O’Flynn. Vol. 12804. Lecture Notes in Computer Science. Halifax, NS, Canada (Virtual Event), Oct. 2020, pp. 404–430. DOI: 10.1007/978-3-030-81652-0_16 (cit. on p. 102).

[72] Leon Groot Bruinderink et al. “Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme”. In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Santa Barbara, CA, USA, Aug. 2016, pp. 323–345. DOI: 10.1007/978-3-662-53140-2_16 (cit. on p. 110).

[73] David Cash, Eike Kiltz and Victor Shoup. “The Twin Diffie-Hellman Problem and Applications”. In: *Advances in Cryptology – EUROCRYPT 2008*. Ed. by Nigel P. Smart. Vol. 4965. Lecture Notes in Computer Science. Istanbul, Turkey, Apr. 2008, pp. 127–145. DOI: 10.1007/978-3-540-78967-3_8 (cit. on pp. 106, 110, 112).

[74] Wouter Castryck and Thomas Decru. “An Efficient Key Recovery Attack on SIDH”. In: *Advances in Cryptology – EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Lyon, France, Apr. 2023, pp. 423–447. DOI: 10.1007/978-3-031-30589-4_15 (cit. on p. 103).

[75] Wouter Castryck and Thomas Decru. *An efficient key recovery attack on SIDH (preliminary version)*. Cryptology ePrint Archive, Report 2022/975. https://eprint.iacr.org/2022/975. 2022 (cit. on p. 120).

[76] Wouter Castryck et al. "CSIDH: An Efficient Post-Quantum Commutative Group Action". In: *Advances in Cryptology – ASIACRYPT 2018, Part III*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11274. Lecture Notes in Computer Science. Brisbane, Queensland, Australia, Dec. 2018, pp. 395–427. DOI: 10.1007/978-3-030-03332-3_15 (cit. on pp. 102, 103).

[77] Melissa Chase et al. "Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1825–1842. DOI: 10.1145/3133956.3133997 (cit. on p. 120).

[78] Graeme Connell and Rolfe Schmidt. *Signal Protocol and Post-Quantum Ratchets*. Signal Blog. https://signal.org/blog/spqr/ (accessed 2025-10-02). 2025 (cit. on p. 2).

[79] Deirdre Connolly, Peter Schwabe and Bas Westerbaan. *X-Wing: general-purpose hybrid post-quantum KEM*. Internet-Draft draft-connolly-cfrg-xwing-kem-06. Work in Progress. Internet Engineering Task Force, Oct. 2024. 34 pp. URL: https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/06/ (cit. on p. 132).

[80] Craig Costello, Patrick Longa and Michael Naehrig. "Efficient Algorithms for Supersingular Isogeny Diffie-Hellman". In: *Advances in Cryptology – CRYPTO 2016, Part I*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9814. Lecture Notes in Computer Science. Santa Barbara, CA, USA, Aug. 2016, pp. 572–601. DOI: 10.1007/978-3-662-53018-4_21 (cit. on p. 103).

[81] Cas Cremers, Caroline Fontaine and Charlie Jacomme. "A Logic and an Interactive Prover for the Computational Post-Quantum Security of Protocols". In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022, pp. 125–141. DOI: 10.1109/SP46214.2022.9833800 (cit. on p. 69).

[82] Jan-Pieter D'Anvers et al. *SABER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020 (cit. on p. 67).

[83] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638 (cit. on pp. 1, 100).

[84] Jintai Ding, Xiang Xie and Xiaodong Lin. *A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem*. Cryptology ePrint Archive, Report 2012/688. https://eprint.iacr.org/2012/688. 2012 (cit. on p. 102).

[85] Samuel Dobson and Steven D. Galbraith. "Post-Quantum Signal Key Agreement from SIDH". In: *Post-Quantum Cryptography*. Ed. by Jung Hee Cheon and Thomas Johansson. Cham: Springer International Publishing, 2022, pp. 422–450. ISBN: 978-3-031-17234-2 (cit. on p. 102).

[86] Jack Doerner et al. "Secure Two-party Threshold ECDSA from ECDSA Assumptions". In: *2018 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2018, pp. 980–997. DOI: 10.1109/SP.2018.00036 (cit. on p. 120).

[87] Jack Doerner et al. "Threshold ECDSA from ECDSA Assumptions: The Multiparty Case". In: *2019 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2019, pp. 1051–1066. DOI: 10.1109/SP.2019.00024 (cit. on p. 120).

[88] Léo Ducas. "Shortest Vector from Lattice Sieving: A Few Dimensions for Free". In: *Advances in Cryptology – EUROCRYPT 2018, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. Lecture Notes in Computer Science. Tel Aviv, Israel: Springer, Cham, Switzerland, Apr. 2018, pp. 125–145. DOI: 10.1007/978-3-319-78381-9_5 (cit. on p. 68).

[89] Léo Ducas and Ludo N. Pulles. "Does the Dual-Sieve Attack on Learning with Errors Even Work?" In: *Advances in Cryptology – CRYPTO 2023, Part III*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14083. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2023, pp. 37–69. DOI: 10.1007/978-3-031-38548-3_2 (cit. on p. 68).

[90] Léo Ducas et al. "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.1 (2018), pp. 238–268. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i1.238-268. URL: https://tches.iacr.org/index.php/TCHES/article/view/839 (cit. on p. 30).

[91] Julien Duman et al. "Faster Lattice-Based KEMs via a Generic Fujisaki-Okamoto Transform Using Prefix Hashing". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 2021, pp. 2722–2737. DOI: 10.1145/3460120.3484819 (cit. on p. 68).

[92] Julien Duman et al. "Group Action Key Encapsulation and Non-Interactive Key Exchange in the QROM". In: *Advances in Cryptology – ASIACRYPT 2022, Part II*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13792. Lecture Notes in Computer Science. Taipei, Taiwan, Dec. 2022, pp. 36–66. DOI: 10.1007/978-3-031-22966-4_2 (cit. on pp. 108, 125).

[93] Morris J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Tech. rep. National Institute of Standards and Technology, July 2015. DOI: 10.6028/nist.fips.202. URL: https://doi.org/10.6028/nist.fips.202 (cit. on p. 124).

[94] Andres Erbsen et al. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises". In: *2019 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2019, pp. 1202–1219. DOI: 10.1109/SP.2019.00005 (cit. on pp. 2, 31, 131).

[95] Thomas Espitau et al. "Side-Channel Attacks on BLISS Lattice-Based Signatures: Exploiting Branch Tracing against strongSwan and Electromagnetic Emanations in Microcontrollers". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1857–1874. DOI: 10.1145/3133956.3134028 (cit. on p. 110).

[96] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology – CRYPTO'86*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Santa Barbara, CA, USA, Aug. 1987, pp. 186–194. DOI: 10.1007/3-540-47721-7_12 (cit. on pp. 102, 194).

[97] Pierre-Alain Fouque et al. *Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU (Specification v1.2)*. Round-3 submission to the NIST PQC standardization project. https://falcon-sign.info/falcon.pdf. 2020 (cit. on p. 30).

[98] Eduarda S. V. Freire et al. "Non-Interactive Key Exchange". In: *PKC 2013: 16th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Kaoru Kurosawa and Goichiro Hanaoka. Vol. 7778. Lecture Notes in Computer Science. Nara, Japan, Feb. 2013, pp. 254–271. DOI: 10.1007/978-3-642-36362-7_17 (cit. on pp. 15, 100, 106, 110–113).

[99] Yu-Fu Fu et al. "Signed Cryptographic Program Verification with Typed CryptoLine". In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Ed. by Lorenzo Cavallaro et al. London, UK: ACM Press, Nov. 2019, pp. 1591–1606. DOI: 10.1145/3319535.3354199 (cit. on pp. 17, 128, 134, 142, 202).

[100] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34 (cit. on pp. 14, 72, 82, 102, 103).

[101] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Journal of Cryptology* 26.1 (Jan. 2013), pp. 80–101. DOI: 10.1007/s00145-011-9114-1 (cit. on pp. 14, 72, 82, 102).

[102] Phillip Gajland et al. "SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange". In: *USENIX Security 2024: 33rd USENIX Security Symposium*. Ed. by Davide Balzarotti and Wenyuan Xu. Philadelphia, PA, USA: USENIX Association, Aug. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/gajland (cit. on pp. 4, 99, 208).

[103] Steven D. Galbraith et al. "On the Security of Supersingular Isogeny Cryptosystems". In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam, Dec. 2016, pp. 63–91. DOI: 10.1007/978-3-662-53887-6_3 (cit. on pp. 103, 120).

[104] Rosario Gennaro and Steven Goldfeder. "Fast Multiparty Threshold ECDSA with Fast Trustless Setup". In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie et al. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 1179–1194. DOI: 10.1145/3243734.3243859 (cit. on p. 120).

[105] Craig Gentry, Chris Peikert and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions". In: *40th Annual ACM Symposium on Theory of Computing*. Ed. by Richard E. Ladner and Cynthia Dwork. Victoria, BC, Canada: ACM Press, May 2008, pp. 197–206. DOI: 10.1145/1374376.1374407 (cit. on pp. 68, 69).

[106] Guillaume Girol et al. "A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols". In: *USENIX Security 2020: 29th USENIX Security Symposium*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, Aug. 2020, pp. 1857–1874 (cit. on p. 2).

[107] Shafi Goldwasser, Silvio Micali and Charles Rackoff. "The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)". In: *17th Annual ACM Symposium on Theory of Computing*. Providence, RI, USA: ACM Press, May 1985, pp. 291–304. DOI: 10.1145/22145.22178 (cit. on pp. 194, 197).

[108] Paul Grubbs, Varun Maram and Kenneth G. Paterson. "Anonymous, Robust Post-quantum Public Key Encryption". In: *Advances in Cryptology – EUROCRYPT 2022, Part III*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13277. Lecture Notes in Computer Science. Trondheim, Norway: Springer, Cham, Switzerland, May 2022, pp. 402–432. DOI: 10.1007/978-3-031-07082-2_15 (cit. on pp. 18, 67, 73).

[109] Shay Gueron and Fabian Schlieker. "Speeding up R-LWE Post-quantum Key Exchange". In: *Secure IT Systems*. Ed. by Billy Bob Brumley and Juha Röning. Vol. 10014. LNCS. https://eprint.iacr.org/2016/467. Springer, 2016, pp. 187–198 (cit. on p. 51).

[110] Qian Guo and Thomas Johansson. "Faster Dual Lattice Attacks for Solving LWE with Applications to CRYSTALS". In: *Advances in Cryptology – ASIACRYPT 2021, Part IV*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13093. Lecture Notes in Computer Science. Singapore: Springer, Cham, Switzerland, Dec. 2021, pp. 33–62. DOI: 10.1007/978-3-030-92068-5_2 (cit. on p. 68).

[111] Qian Guo, Thomas Johansson and Alexander Nilsson. "A Key-Recovery Timing Attack on Post-quantum Primitives Using the Fujisaki-Okamoto Transformation and Its Application on FrodoKEM". In: *Advances in Cryptology – CRYPTO 2020, Part II*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2020, pp. 359–386. DOI: 10.1007/978-3-030-56880-1_13 (cit. on p. 68).

[112] Siyao Guo et al. "Limits on the Efficiency of (Ring) LWE Based Non-interactive Key Exchange". In: *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Aggelos Kiayias et al. Vol. 12110. Lecture Notes in Computer Science. Edinburgh, UK, May 2020, pp. 374–395. DOI: 10.1007/978-3-030-45374-9_13 (cit. on pp. 100, 104).

[113] Mike Hamburg et al. "Chosen Ciphertext k-Trace Attacks on Masked CCA2 Secure Kyber". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (2021), pp. 88–113. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.88-113. URL: https://tches.iacr.org/index.php/TCHES/article/view/9061 (cit. on p. 68).

[114] Keitaro Hashimoto et al. "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable". In: *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Juan Garay. Vol. 12711. Lecture Notes in Computer Science. Virtual Event, May 2021, pp. 410–440. DOI: 10.1007/978-3-030-75248-4_15 (cit. on p. 102).

[115] Julius Hermelink et al. "Adapting Belief Propagation to Counter Shuffling of NTTs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.1 (2023), pp. 60–88. DOI: 10.46586/tches.v2023.i1.60-88 (cit. on p. 68).

[116] Julius Hermelink et al. "Belief Propagation Meets Lattice Reduction: Security Estimates for Error-Tolerant Key Recovery from Decryption Errors". In: *IACR Transactions on Cryptographic Hardware and*

Embedded Systems 2023.4 (2023), pp. 287–317. DOI: 10.46586/tches.v2023.i4.287-317 (cit. on p. 68).

[117] Dennis Hofheinz, Kathrin Hövelmanns and Eike Kiltz. *A Modular Analysis of the Fujisaki-Okamoto Transformation*. Cryptology ePrint Archive, Report 2017/604. 2017. URL: https://eprint.iacr.org/2017/604 (cit. on pp. 79, 82–87, 89, 187).

[118] Dennis Hofheinz, Kathrin Hövelmanns and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *TCC 2017: 15th Theory of Cryptography Conference, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Lecture Notes in Computer Science. Baltimore, MD, USA: Springer, Cham, Switzerland, Nov. 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12 (cit. on pp. 72, 73, 79, 82–87, 89, 187).

[119] Kathrin Hövelmanns, Andreas Hülsing and Christian Majenz. "Failing Gracefully: Decryption Failures and the Fujisaki-Okamoto Transform". In: *Advances in Cryptology – ASIACRYPT 2022, Part IV*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13794. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Dec. 2022, pp. 414–443. DOI: 10.1007/978-3-031-22972-5_15 (cit. on p. 68).

[120] Kathrin Hövelmanns et al. "Generic Authenticated Key Exchange in the Quantum Random Oracle Model". In: *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Aggelos Kiayias et al. Vol. 12111. Lecture Notes in Computer Science. Edinburgh, UK: Springer, Cham, Switzerland, May 2020, pp. 389–422. DOI: 10.1007/978-3-030-45388-6_14 (cit. on p. 83).

[121] Andreas Hülsing and Mikhail A. Kudinov. "Recovering the Tight Security Proof of SPHINCS$^+$". In: *Advances in Cryptology – ASIACRYPT 2022, Part IV*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13794. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Dec. 2022, pp. 3–33. DOI: 10.1007/978-3-031-22972-5_1 (cit. on p. 17).

[122] Andreas Hülsing, Matthias Meijers and Pierre-Yves Strub. "Formal Verification of Saber's Public-Key Encryption Scheme in EasyCrypt". In: *Advances in Cryptology – CRYPTO 2022, Part I*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13507. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2022, pp. 622–653. DOI: 10.1007/978-3-031-15802-5_22 (cit. on pp. 35, 66, 68, 76).

[123] Andreas Hülsing, Joost Rijneveld and Fang Song. "Mitigating Multi-target Attacks in Hash-Based Signatures". In: *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Chen-Mou Cheng et al. Vol. 9614. Lecture

Notes in Computer Science. Taipei, Taiwan: Springer Berlin Heidelberg, Germany, Mar. 2016, pp. 387–416. DOI: 10.1007/978-3-662-49384-7_15 (cit. on p. 17).

[124] Andreas Hülsing et al. "Post-quantum WireGuard". In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2021, pp. 304–321. DOI: 10.1109/SP40001.2021.00030 (cit. on p. 101).

[125] Vincent Hwang et al. "Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.4 (2022). https://tches.iacr.org/index.php/TCHES/article/view/9838, pp. 718–750 (cit. on pp. 35, 41, 61).

[126] David Jao and Luca De Feo. "Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies". In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*. Ed. by Bo-Yin Yang. Tapei, Taiwan, Nov. 2011, pp. 19–34. DOI: 10.1007/978-3-642-25405-5_2 (cit. on pp. 103, 120).

[127] David Jao et al. *SIKE*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions. National Institute of Standards and Technology, 2017 (cit. on p. 103).

[128] Haodong Jiang et al. "IND-CCA-Secure Key Encapsulation Mechanism in the Quantum Random Oracle Model, Revisited". In: *Advances in Cryptology – CRYPTO 2018, Part III*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10993. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2018, pp. 96–125. DOI: 10.1007/978-3-319-96878-0_4 (cit. on p. 68).

[129] José Bacelar Almeida and Manuel Barbosa and Gilles Barthe and Benjamin Grégoire and Vincent Laporte and Jean-Christophe Léchenet and Tiago Oliveira and Hugo Pacheco and Miguel Quaresma and Peter Schwabe and Antoine Séré and Pierre-Yves Strub. "Formally verifying Kyber – Episode IV: Implementation correctness". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (June 2023), pp. 164–193. DOI: 10.46586/tches.v2023.i3.164-193 (cit. on pp. 3, 30, 66–68, 91, 92, 96, 97, 128, 132, 150, 153, 155, 207).

[130] Huang Junhao. *Questions about the range analysis of iNTT for "Faster Kyber and Dilithium on the Cortex-M4"*. https://github.com/mupq/pqm4/issues/226 (accessed 2025-10-11). 2022 (cit. on p. 18).

[131] Matthias J. Kannwischer et al. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. https://github.com/mupq/pqm4 (cit. on p. 18).

[132] John Kelsey, Shu-jen Change and Ray Perlner. *SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash*. Tech. rep. National Institute of Standards and Technology, Dec. 2016. DOI: 10.6028/nist.sp.800-185. URL: https://doi.org/10.6028/nist.sp.800-185 (cit. on pp. 123, 124).

[133] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. https://doi.org/10.1109/SP.2019.00002. IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002. URL: https://doi.org/10.1109/SP.2019.00002 (cit. on p. 34).

[134] Bor de Kock. "A non-interactive key exchange based on ring-learning with errors". MA thesis. Master's thesis, Eindhoven University of Technology, 2018 (cit. on p. 103).

[135] Hugo Krawczyk and Hoeteck Wee. "The OPTLS Protocol and TLS 1.3". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. Saarbruecken, Germany: IEEE, Mar. 2016, pp. 81–96. DOI: 10.1109/eurosp.2016.18. URL: https://doi.org/10.1109/eurosp.2016.18 (cit. on p. 101).

[136] Katharina Kreuzer. *Verification of Correctness and Security Properties for CRYSTALS-KYBER*. Cryptology ePrint Archive, Report 2023/087. 2023. URL: https://eprint.iacr.org/2023/087 (cit. on pp. 68, 76, 79).

[137] Katharina Kreuzer. *Verification of the $(1-\delta)$-Correctness Proof of CRYSTALS-KYBER with Number Theoretic Transform*. IACR ePrint Archive report 2023/27. https://eprint.iacr.org/2023/27. 2023 (cit. on pp. 35, 41).

[138] Mikhail Kudinov, Evgeniy Kiktenko and Aleksey Fedorov. *ROUND 3 OFFICIAL COMMENT: SPHINCS$^+$*. pqc-forum. https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/official-comments/Sphincs-Plus-round3-official-comment.pdf (cit. on p. 17).

[139] Joel Kuepper et al. "CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1268–1292. DOI: 10.1145/3591272. URL: https://doi.org/10.1145/3591272 (cit. on p. 131).

[140] Kris Kwiatkowski and Luke Valenta. *The TLS Post-Quantum Experiment*. Post on the Cloudflare blog. https://blog.cloudflare.com/the-tls-post-quantum-experiment/. 2019 (cit. on pp. 31, 101).

[141] Adam Langley. *CECPQ1 results*. Blog post. https://www.imperialviolet.org/2016/11/28/cecpq1.html. 2016 (cit. on pp. 31, 101).

[142] Adam Langley. *CECPQ2*. Blog post. https://www.imperialviolet.org/2018/12/12/cecpq2.html. 2018 (cit. on pp. 31, 101).

[143] Adam Langley, Mike Hamburg and Sean Turner. *Elliptic Curves for Security*. RFC 7748. Jan. 2016. DOI: 10.17487/RFC7748. URL: https://www.rfc-editor.org/info/rfc7748 (cit. on p. 132).

[144] Adeline Langlois and Damien Stehlé. "Worst-Case to Average-Case Reductions for Module Lattices". In: *Designs, Codes and Cryptography* 75.3 (2015), pp. 565–599 (cit. on pp. 6, 11, 109).

[145] Yehuda Lindell and Ariel Nof. "Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody". In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie et al. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 1837–1854. DOI: 10.1145/3243734.3243788 (cit. on p. 120).

[146] Felix Linker, Ralf Sasse and David Basin. *A Formal Analysis of Apple's iMessage PQ3 Protocol*. Cryptology ePrint Archive, Paper 2024/1395. 2024. URL: https://eprint.iacr.org/2024/1395 (cit. on p. 2).

[147] Vadim Lyubashevsky. *Converting NewHope/LWE key exchange to a Diffe-Hellman-like algorithm*. Crypto Stack Exchange. [Online:] https://crypto.stackexchange.com/questions/48146/converting-newhope-lwe-key-exchange-to-a-diffe-hellman-like-algorithm. 2017. URL: https://crypto.stackexchange.com/questions/48146/converting-newhope-lwe-key-exchange-to-a-diffe-hellman-like-algorithm (visited on 10/06/2017) (cit. on pp. 102, 103).

[148] Vadim Lyubashevsky, Ngoc Khanh Nguyen and Maxime Plançon. "Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General". In: *Advances in Cryptology – CRYPTO 2022, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. Lecture Notes in Computer Science. Santa Barbara, CA, USA, Aug. 2022, pp. 71–101. DOI: 10.1007/978-3-031-15979-4_3 (cit. on pp. 103, 120, 122).

[149] Vadim Lyubashevsky, Ngoc Khanh Nguyen and Gregor Seiler. "Practical Lattice-Based Zero-Knowledge Proofs for Integer Relations". In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti et al. Virtual Event, USA: ACM Press, Nov. 2020, pp. 1051–1070. DOI: 10.1145/3372297.3417894 (cit. on pp. 103, 120).

[150] Vadim Lyubashevsky, Chris Peikert and Oded Regev. *On Ideal Lattices and Learning With Errors Over Rings*. Slides of the talk given by Chris Peikert at Eurocrypt 2010. https://iacr.org/conferences/eurocrypt2010/talks/slides-ideal-lwe.pdf. 2010 (cit. on p. 72).

[151] Vadim Lyubashevsky, Chris Peikert and Oded Regev. *On Ideal Lattices and Learning with Errors Over Rings*. Cryptology ePrint Archive, Report 2012/230. 2012. URL: https://eprint.iacr.org/2012/230 (cit. on p. 77).

[152] Vadim Lyubashevsky, Chris Peikert and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. French Riviera: Springer Berlin Heidelberg, Germany, May 2010, pp. 1–23. DOI: 10.1007/978-3-642-13190-5_1 (cit. on pp. 6, 11, 72, 77).

[153] Vadim Lyubashevsky et al. *CRYSTALS-DILITHIUM*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 100).

[154] Vadim Lyubashevsky et al. *CRYSTALS-DILITHIUM: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project [170]. https://pq-crystals.org/dilithium. 2017 (cit. on p. 2).

[155] Luciano Maino et al. "A Direct Key Recovery Attack on SIDH". In: *Advances in Cryptology – EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Lyon, France, Apr. 2023, pp. 448–471. DOI: 10.1007/978-3-031-30589-4_16 (cit. on p. 103).

[156] Giulio Malavolta et al. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability". In: *ISOC Network and Distributed System Security Symposium – NDSS 2019*. San Diego, CA, USA: The Internet Society, Feb. 2019 (cit. on p. 120).

[157] Varun Maram and Keita Xagawa. "Post-quantum Anonymity of Kyber". In: *PKC 2023: 26th International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Alexandra Boldyreva and Vladimir Kolesnikov. Vol. 13940. Lecture Notes in Computer Science. Atlanta, GA, USA: Springer, Cham, Switzerland, May 2023, pp. 3–35. DOI: 10.1007/978-3-031-31368-4_1 (cit. on pp. 68, 73).

[158] Moxie Marlinspike and Trevor Perrin. *The Double Ratchet Algorithm*. 2016. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf (cit. on p. 100).

[159] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol (Revision 1)*. Part of the Signal Protocol Documentation. https://signal.org/docs/specifications/x3dh/x3dh.pdf. 2016 (cit. on pp. 100, 101).

[160] Simon Meier et al. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*. CAV 2013. Saint Petersburg, Russia: Springer-Verlag, 2013, pp. 696–701. ISBN: 9783642397981 (cit. on pp. 17, 160).

[161] Daniele Micciancio and Chris Peikert. “Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller”. In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Cambridge, UK: Springer Berlin Heidelberg, Germany, Apr. 2012, pp. 700–718. DOI: 10.1007/978-3-642-29011-4_41 (cit. on p. 68).

[162] Daniele Micciancio and Panagiotis Voulgaris. “Faster exponential time algorithms for the shortest vector problem”. In: *SODA ’10: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2010, pp. 1468–1480 (cit. on p. 68).

[163] Peter L. Montgomery. “Modular Multiplication without Trial Division”. In: *Mathematics of Computation* 44.170 (1985), pp. 519–521 (cit. on p. 124).

[164] Kaushik Nath and Daniel J. Bernstein. *lib25519*. https://lib25519.cr.yp.to/. 2022 (cit. on p. 126).

[165] Kalle Ngo et al. “A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM Implementation”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (2021), pp. 676–707. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.676-707. URL: https://tches.iacr.org/index.php/TCHES/article/view/9079 (cit. on p. 68).

[166] Phong Q. Nguyen and Thomas Vidick. “Sieve algorithms for the shortest vector problemare practical”. In: *J. Math. Crypt.* (2 2008). https://doi.org/10.1515/JMC.2008.009, pp. 181–207 (cit. on p. 68).

[167] National Institute of Standards and Technology. *FIPS PUB 203 (Initial Public Draft) – Module-Lattice-based Key-Encapsulation Mechanism Standard*. https://csrc.nist.gov/pubs/fips/203/ipd. 2023 (cit. on pp. 2, 3, 12, 65, 73, 75, 91, 132, 153).

[168] *FIPS PUB 202 – SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. National Institute of Standards and Technology, 2015 (cit. on p. 74).

[169] NIST. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. 2016 (cit. on p. 121).

[170] NIST Computer Security Division. *Post-Quantum Cryptography Standardization*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. 2016 (cit. on pp. 6, 165, 180, 182).

[171] OpenSSH. *OpenSSH 9.0 Release Notes*. https://www.openssh.com/txt/release-9.0. 2022 (cit. on p. 31).

[172] Christian Paquin, Douglas Stebila and Goutam Tamvada. "Benchmarking Post-quantum Cryptography in TLS". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Paris, France, Apr. 2020, pp. 72–91. DOI: 10.1007/978-3-030-44223-1_5 (cit. on p. 101).

[173] Chris Peikert. "He Gives C-Sieves on the CSIDH". In: *Advances in Cryptology – EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Zagreb, Croatia, May 2020, pp. 463–492. DOI: 10.1007/978-3-030-45724-2_16 (cit. on p. 103).

[174] Trevor Perrin. *Noise Protocol Framework*. https://noiseprotocol.org/noise.pdf (Revision 34 vom 2018-07-11). 2018 (cit. on p. 100).

[175] Peter Pessl, Leon Groot Bruinderink and Yuval Yarom. "To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1843–1855. DOI: 10.1145/3133956.3134023 (cit. on p. 110).

[176] Peter Pessl and Lukas Prokop. "Fault Attacks on CCA-secure Lattice KEMs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (2021), pp. 37–60. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i2.37-60. URL: https://tches.iacr.org/index.php/TCHES/article/view/8787 (cit. on p. 68).

[177] Rachel Player. "Parameter selection in lattice-based cryptography". PhD thesis. Royal Holloway, University of London, 2018 (cit. on p. 120).

[178] Marina Polubelova et al. "HACLxN: Verified generic SIMD crypto (for all your favourite platforms)". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 899–918 (cit. on p. 2).

[179] Thomas Prest et al. *FALCON*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on p. 100).

[180] Thomas Prest et al. *Falcon: Algorithm Specification and Supporting Documentation*. Submission to the NIST Post-Quantum Cryptography Standardization Project [170]. https://falcon-sign.info/. 2017 (cit. on p. 2).

[181] Robert Primas, Peter Pessl and Stefan Mangard. "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Sept. 2017, pp. 513–533. DOI: 10.1007/978-3-319-66787-4_25 (cit. on p. 68).

[182] Oded Regev. ""On lattices, learning with errors, random linear codes, and cryptography"". In: *37th Annual ACM Symposium on Theory of Computing*. Ed. by Harold N. Gabow and Ronald Fagin. Baltimore, MA, USA: ACM Press, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603 (cit. on pp. 6–9, 100, 109).

[183] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://rfc-editor.org/rfc/rfc8446.txt (cit. on p. 100).

[184] Ronald L. Rivest, Adi Shamir and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the Association for Computing Machinery* 21.2 (Feb. 1978), pp. 120–126. DOI: 10.1145/359340.359342 (cit. on p. 1).

[185] Damien Robert. "Breaking SIDH in Polynomial Time". In: *Advances in Cryptology – EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Lyon, France, Apr. 2023, pp. 472–503. DOI: 10.1007/978-3-031-30589-4_17 (cit. on p. 103).

[186] Tsunekazu Saito, Keita Xagawa and Takashi Yamakawa. *Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model*. Cryptology ePrint Archive, Report 2017/1005. 2017. URL: https://eprint.iacr.org/2017/1005 (cit. on p. 72).

[187] Tsunekazu Saito, Keita Xagawa and Takashi Yamakawa. "Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model". In: *Advances in Cryptology – EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. Lecture Notes in Computer Science. Tel Aviv, Israel: Springer, Cham, Switzerland, Apr. 2018, pp. 520–551. DOI: 10.1007/978-3-319-78372-7_17 (cit. on p. 68).

[188] Benedikt Schmidt et al. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: CSF '12. USA: IEEE Computer Society, 2012, pp. 78–94. ISBN: 9780769547183. DOI: 10.1109/CSF.2012.25. URL: https://doi.org/10.1109/CSF.2012.25 (cit. on pp. 17, 160).

[189] Peter Schwabe, Douglas Stebila and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures". In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti et al. Virtual Event, USA: ACM Press, Nov. 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350 (cit. on p. 101).

[190] Peter Schwabe et al. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions. National Institute of Standards and Technology, 2017 (cit. on p. 67).

[191] Peter Schwabe et al. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022 (cit. on pp. 68, 100, 110, 122, 123, 125).

[192] Gregor Seiler. *Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography*. Cryptology ePrint Archive, Report 2018/039. https://eprint.iacr.org/2018/039. 2018 (cit. on p. 123).

[193] Basavesh Ammanaghatta Shivakumar et al. "Enforcing Fine-grained Constant-time Policies". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin et al. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 83–96. DOI: 10.1145/3548606.3560689 (cit. on p. 20).

[194] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science*. Santa Fe, NM, USA: IEEE Computer Society Press, Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700 (cit. on pp. 1, 100).

[195] Sara Stadler et al. *Hybrid Signal protocol for post-quantum email encryption*. Cryptology ePrint Archive, Report 2021/875. https://eprint.iacr.org/2021/875. 2021 (cit. on p. 102).

[196] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655. URL: https://doi.org/10.1145/2837614.2837655 (cit. on p. 17).

[197] Ehsan Ebrahimi Targhi and Dominique Unruh. "Post-Quantum Security of the Fujisaki-Okamoto and OAEP Transforms". In: *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*. Ed. by Martin Hirt and Adam D. Smith. Vol. 9986. Lecture Notes in Computer Science. Beijing, China: Springer Berlin Heidelberg, Germany, Oct. 2016, pp. 192–216. DOI: 10.1007/978-3-662-53644-5_8 (cit. on p. 68).

[198] Sri Aravinda Krishnan Thyagarajan et al. "Efficient CCA Timed Commitments in Class Groups". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 2021, pp. 2663–2684. DOI: 10.1145/3460120.3484773 (cit. on p. 120).

[199] Ming-Hsien Tsai, Bow-Yaw Wang and Bo-Yin Yang. "Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1973–1987. DOI: 10.1145/3133956.3134076 (cit. on pp. 17, 128, 131, 134, 142, 202).

[200] Dominique Unruh. "Non-Interactive Zero-Knowledge Proofs in the Quantum Random Oracle Model". In: *Advances in Cryptology – EUROCRYPT 2015, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. Lecture Notes in Computer Science. Sofia, Bulgaria, Apr. 2015, pp. 755–784. DOI: 10.1007/978-3-662-46803-6_25 (cit. on pp. 120, 195–197).

[201] Dominique Unruh. "Post-Quantum Verification of Fujisaki-Okamoto". In: *Advances in Cryptology – ASIACRYPT 2020, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Daejeon, South Korea: Springer, Cham, Switzerland, Dec. 2020, pp. 321–352. DOI: 10.1007/978-3-030-64837-4_11 (cit. on pp. 35, 69).

[202] Dominique Unruh. "Quantum relational Hoare logic". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 33:1–33:31. DOI: 10.1145/3290346. URL: https://doi.org/10.1145/3290346 (cit. on p. 69).

[203] Viktoriya V. Vysotskaya and Lev I. Vysotsky. "Invertible matrices over some quotient rings: identification, generation, and analysis". In: *Discrete Mathematics and Applications* 32.4 (2022), pp. 263–278. DOI: doi:10.1515/dma-2022-0022. URL: https://doi.org/10.1515/dma-2022-0022 (cit. on p. 193).

[204] Bas Westerbaan and Cefan Daniel Rubin. *Defending against future threats: Cloudflare goes post-quantum*. Post on the Cloudflare blog. https://blog.cloudflare.com/post-quantum-for-all/. 2019 (cit. on p. 101).

[205] Katherine Q. Ye et al. "Verified Correctness and Security of mbedTLS HMAC-DRBG". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 2007–2020. DOI: 10.1145/3133956.3133974 (cit. on p. 68).

[206] Mark Zhandry. "How to Record Quantum Queries, and Applications to Quantum Indifferentiability". In: *Advances in Cryptology – CRYPTO 2019, Part II*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11693. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2019, pp. 239–268. DOI: 10.1007/978-3-030-26951-7_9 (cit. on p. 68).

[207] Yi Zhou et al. "Galápagos: Developing Verified Low Level Cryptography on Heterogeneous Hardwares". In: *ACM CCS 2023: 30th Conference on Computer and Communications Security*. Ed. by Weizhi Meng et al. Copenhagen, Denmark: ACM Press, Nov. 2023, pp. 2113–2127. DOI: 10.1145/3576915.3616603 (cit. on p. 131).

[208] Jean Karim Zinzindohoué et al. "HACL*: A Verified Modern Cryptographic Library". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1789–1806. DOI: 10.1145/3133956.3134043 (cit. on pp. 31, 131).

A

Formally verifying Kyber: Episode V

A.1 Cryptographic Definitions

For a set S, $|S|$ denotes the cardinality of S. For a finite set S, we denote the sampling of a uniform random element x by $x \leftarrow\!\!\$\, S$. We denote deterministic computation of an algorithm A on input x by $y \leftarrow A(x)$. We denote algorithms with access to an oracle O by A^O. Unless stated otherwise, we assume all our algorithms to be probabilistic and denote the computation by $y \leftarrow\!\!\$\, A(x)$. When deterministically computing a randomized algorithms by fixing its random coins r, we write: $y \leftarrow A(x; r)$. For the most part of this section we follow [118, 117].

A.1.0.1 Pseudorandomness.

Let $\mathsf{f} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ be a function. We define the PRF advantage of an adversary $\mathcal{A}$ as

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{f}}(\mathcal{A}) = \left| \Pr[\mathcal{A}^{\mathsf{PRF}(K,\cdot)} \Rightarrow 1 : K \leftarrow\!\!\$\, \mathcal{K}] - \Pr[\mathcal{A}^{\mathsf{F}(\cdot)} \Rightarrow 1 : \mathsf{F} \leftarrow\!\!\$\, (\mathcal{X} \to \mathcal{Y})] \right| .$$

When this advantage is small for all efficient adversaries, we say f is a pseudorandom function. Let $\mathsf{g} : \mathcal{K} \to \mathcal{Y}$ be a function. We define the PRG advantage of an adversary $\mathcal{A}$ as

$$\mathsf{Adv}^{\mathsf{PRG}}_{\mathsf{g}}(\mathcal{A}) = \left| \Pr[\mathcal{A}(y) \Rightarrow 1 : K \leftarrow\!\!\$\, \mathcal{K}; y \leftarrow \mathsf{g}(K)] - \Pr[\mathcal{A}(y) \Rightarrow 1 : y \leftarrow\!\!\$\, \mathcal{Y}] \right| .$$

When this advantage is small for all efficient adversaries, we say g is a pseudorandom generator.

A.1.0.2 Public-Key Encryption.

A public-key encryption scheme consists of three algorithms $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and a finite message space $\mathcal{M}$. The key generation algorithm Gen outputs a key pair (pk, sk). The encryption algorithm Enc, on input pk and a message $m \in \mathcal{M}$, outputs a ciphertext $c \leftarrow\!\!\$\, \mathsf{Enc}(pk, m)$. If necessary to derandomize encryption, we writing $c \leftarrow \mathsf{Enc}(pk, m; r)$, where $r \leftarrow\!\!\$\, \mathcal{R}$ and $\mathcal{R}$

Game COR:	Game COR-RO:
1: $(pk, sk) \leftarrow_\$ \mathsf{Gen}(\,)$	1: $(pk, sk) \leftarrow_\$ \mathsf{Gen}(\,)$
2: $m \leftarrow_\$ \mathcal{A}(pk, sk)$	2: $m \leftarrow_\$ \mathcal{A}^{\mathsf{G}}(pk, sk)$
3: $c \leftarrow_\$ \mathsf{Enc}(pk, m)$	3: $c \leftarrow_\$ \mathsf{Enc}^{\mathsf{G}}(pk, m)$
4: **return** $(m \neq \mathsf{Dec}(sk, c))$	4: **return** $(m \neq \mathsf{Dec}^{\mathsf{G}}(sk, c))$

Figure A.1: Correctness of a PKE in the standard model and ROM.

is the randomness space. The decryption algorithm Dec, on input sk and a ciphertext c, outputs either a message $m \leftarrow \mathsf{Dec}(sk, c) \in \mathcal{M}$ or a special symbol $\perp \notin \mathcal{M}$ to indicate that c is not a valid ciphertext.

Correctness. Correctness of a PKE is defined as in Figure A.1. We give both a standard model definition (left) and a definition in the random oracle model (ROM, right). Note that the adversary gets the secret key as an input. In the standard model, δ-correctness means that for all (possibly unbounded) adversaries $\mathcal{A}$, $\Pr[\mathsf{COR}^{\mathcal{A}}_{\mathsf{PKE}} \Rightarrow 1] \leq \delta$. In the ROM, we extend the definition so that δ is parameterized by an upper bound on the number of queries placed by $\mathcal{A}$ to its random oracle.

Ciphertext Spreadness. We say that PKE is γ-spread if, for every key pair $(pk, sk) \leftarrow_\$ \mathsf{Gen}(\,)$ and every ciphertext c, $\Pr_{r \leftarrow_\$ \mathcal{R}}[c = \mathsf{Enc}(pk, m; r)] \leq 2^{-\gamma}$.

Security. We now define several security notions for public-key encryption.

Definition 1 ($\mathsf{OW\text{-}Atk}$). *Let* $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a public-key encryption scheme with message space* $\mathcal{M}$*. For* $\mathsf{Atk} \in \{\mathsf{CPA}, \mathsf{PCVA}\}$*, we give the* $\mathsf{OW\text{-}Atk}$ *security games in Figure A.2. Note that, if* $\mathsf{Atk}{=}\mathsf{CPA}$*, we have* $\mathcal{O} = \varnothing$*; and if* $\mathsf{Atk}{=}\mathsf{PCVA}$*, we have* $\mathcal{O} = \{\textsc{Pco}, \textsc{Cvo}\}$*. We define the* $\mathsf{OW\text{-}Atk}$ *advantage function of an adversary* $\mathcal{A}$ *against* PKE *as* $\mathsf{Adv}^{\mathsf{OW\text{-}Atk}}_{\mathsf{PKE}}(\mathcal{A}) = \Pr[\mathsf{OW\text{-}Atk}^{\mathcal{A}}_{\mathsf{PKE}} \Rightarrow 1]$.

Note that the adversary wins the one-way game if it guesses the result of decrypting the challenge ciphertext, rather than the standard definition of finding the message that is originally encrypted. The two games have statistical distance δ, if PKE is δ-correct. We also define the more standard notion, allowing the adversary to output a list of guesses.

Definition 2. $\mathsf{OWL\text{-}CPA}$ Let $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme with message space $\mathcal{M}$. We give the $\mathsf{OWL\text{-}CPA}$ security game in Figure A.2. We define the $\mathsf{OWL\text{-}CPA}$ advantage function of an adversary $\mathcal{A}$ against PKE as $\mathsf{Adv}^{\mathsf{OWL\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) = \Pr[\mathsf{OW\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE}} \Rightarrow 1]$.

We conclude this section with the standard definition of $\mathsf{IND\text{-}CPA}$ security for a PKE.

Game OW-Atk:	Oracle $\textsc{Pco}(m \in \mathcal{M}, c)$:	Game OWL-CPA:
1: $(pk, sk) \hookleftarrow_{\$} \mathsf{Gen}(\,)$	1: **return** $(m = \mathsf{Dec}(sk, c))$	1: $(pk, sk) \hookleftarrow_{\$} \mathsf{Gen}(\,)$
2: $m^* \hookleftarrow_{\$} \mathcal{M}$		2: $m^* \hookleftarrow_{\$} \mathcal{M}$
3: $c^* \hookleftarrow_{\$} \mathsf{Enc}(pk, m^*)$	Oracle $\textsc{Cvo}(c \neq c^*)$:	3: $c^* \hookleftarrow_{\$} \mathsf{Enc}(pk, m^*)$
4: $m' \leftarrow \mathcal{A}^{\mathcal{O}}(pk, c^*)$	1: $m \leftarrow \mathsf{Dec}(sk, c)$	4: $L \leftarrow \mathcal{A}(pk, c^*)$
5: **return** $\textsc{Pco}(m', c^*)$	2: **return** $m \neq \bot$	5: **return** $m^* \in L$

Figure A.2: One-wayness of a PKE.

Game IND-CPA:	Game IND-CCA:	Oracle $\mathsf{Decaps}(c)$:
1: $(pk, sk) \hookleftarrow_{\$} \mathsf{Gen}(\,)$	1: $(pk, sk) \hookleftarrow_{\$} \mathsf{Gen}(\,)$	1: If $c = c^*$ **return** $\bot$
2: $(m_0, m_1, st) \hookleftarrow_{\$} \mathcal{A}_1(pk)$	2: $(k_0, c^*) \hookleftarrow_{\$} \mathsf{Enc}(pk)$	2: $k \leftarrow \mathsf{Dec}(sk, c)$
3: $b \hookleftarrow_{\$} \{0,1\}$	3: $k_1 \hookleftarrow_{\$} \mathcal{K}$	3: **return** k
4: $c^* \hookleftarrow_{\$} \mathsf{Enc}(pk, m_b)$	4: $b \hookleftarrow_{\$} \{0,1\}$	
5: $b' \hookleftarrow_{\$} \mathcal{A}_2(st, c^*)$	5: $b' \hookleftarrow_{\$} \mathcal{A}^{\mathsf{Decaps}}(pk, c^*, k_b)$	
6: **return** $b' = b$	6: **return** $b' = b$	

Figure A.3: IND-CPA security of a PKE and IND-CCA security for a KEM.

Definition 3. IND-CPA Let $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme with message space $\mathcal{M}$. We define the IND-CPA game as in Figure A.3, and the IND-CPA advantage function of an adversary $\mathcal{A} = (\mathcal{A}_1, A_2)$ against PKE as $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) = |\Pr[\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE}} \Rightarrow 1] - 1/2|$.

In the proof of the FO transform (see Section 4.5) we use this general lemma that is verified in EasyCrypt.

Lemma 1. *For all* OWL-CPA *adversary* $\mathcal{B}$ *returning a list of size at most* $\textsc{Max}$*, there exists an* IND-CPA *adversary* $\mathcal{A}$ *such that*

$$\mathsf{Adv}^{\mathsf{OWL-CPA}}_{\mathsf{PKE}}(\mathcal{B}) \leq 2 \cdot \left(\frac{\textsc{Max}}{|\mathcal{M}|} + \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) \right)$$

Sketch of formalization. We define an IND-CPA adversary $\mathcal{A}$ that uses the OWL-CPA adversary $\mathcal{B}$ in the obvious way: i. choose challenge IND-CPA messages (m_0, m_1) uniformly at random and run $\mathcal{B}$ on the challenge ciphertext; ii. if the list output by $\mathcal{B}$ contains m_b, but not $m_{\neg b}$, return b; iii. otherwise return a random bit b'.

We re-express the IND-CPA advantage using the standard form that conditions on the hidden bit, and get that:

$$2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) = \left| \Pr[\mathsf{CPA}^{\mathsf{PKE}}_{\mathcal{A}}(1) \Rightarrow 1] - \Pr[\mathsf{CPA}^{\mathsf{PKE}}_{\mathcal{A}}(0) \Rightarrow 1] \right|$$

We then define a bad event in both experiments on the right-hand side of the above equation, which corresponds to $\mathcal{A}$ returning random bit b'. Clearly, this

event occurs with the same probability in both experiments, and the output of both experiments are identically distributed when this event does occur. This means that

$$2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) = |\Pr[\mathsf{CPA}^{\mathsf{PKE}}_{\mathcal{A}}(1) \Rightarrow 1 \wedge \neg\mathsf{bad}] - \Pr[\mathsf{CPA}^{\mathsf{PKE}}_{\mathcal{A}}(0) \Rightarrow 1 \wedge \neg\mathsf{bad}]|$$

The next step in the proof shows that

$$\Pr[\mathsf{CPA}^{\mathsf{PKE}}_{\mathcal{A}}(0) \Rightarrow 1 \wedge \neg\mathsf{bad}] \leq \textsc{Max}/|\mathcal{M}|\,.$$

which follows from the fact that m_1 is information theoretically hidden from $\mathcal{B}$. A similar argument allows us to show that

$$\left|\mathsf{Adv}^{\mathsf{OWL\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{B}) - \Pr[\mathsf{CPA}^{\mathsf{PKE}}_{\mathcal{A}}(1) \Rightarrow 1 \wedge \neg\mathsf{bad}]\right| \leq \textsc{Max}/|\mathcal{M}|\,.$$

The lemma follows from the above inequalities. The arguments above are also good examples of game-hopping proofs that the EasyCrypt logic was designed to handle. ■

A.1.0.3 Key Encapsulation Mechanisms.

A key encapsulation mechanism $\mathsf{KEM} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ consists of three algorithms. The key generation algorithm Gen outputs a key pair (pk, sk), where pk also defines a finite key space $\mathcal{K}$. The encapsulation algorithm Enc, on input pk, outputs a tuple (k, c) where c is said to be an encapsulation of the key k which is contained in key space $\mathcal{K}$. The deterministic decapsulation algorithm Dec, on input sk and an encapsulation c, outputs either a key $k \leftarrow \mathsf{Dec}(sk, c) \in \mathcal{K}$ or a special symbol $\perp \notin \mathcal{K}$ to indicate that c is not a valid encapsulation. We call KEM δ-correct if $\Pr[\mathsf{Dec}(sk, c) \neq k \,|\, (pk, sk) \leftarrow_{\$} Gen; (k, c) \leftarrow_{\$} \mathsf{Enc}(pk)] \leq \delta$. Note that the above definition does not include a correctness adversary—and for this reason also makes sense in the random oracle model—because KEM ciphertexts do not depend on maliciously chosen messages.

Security. We now define indistinguishability under chosen ciphertext attacks (IND-CCA) for a KEM.

Definition 4. IND-CCA. We define the IND-CCA game also in Figure A.3 and the IND-CCA advantage function of an adversary $\mathcal{A}$ against KEM as

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{A}) = \left|\Pr[\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{KEM}} \Rightarrow 1] - 1/2\right|\,.$$

A.2 Generic EasyCrypt Arguments

We now describe more in depth the generic arguments we make use of in our formal proofs, and some of the issues faced in applying them to the specific ML-KEM proof.

A.2.1 Plug-and-Pray

The first generic result we use heavily in this proof is a *plug-and-pray* argument. In this argument, a parametric reduction R_i—for example, a reduction that simulates the ith query to an oracle in a specific way—is replaced, with some security loss, with a reduction R that 1) makes a uniformly random guess as to the value of the parameter i (within a bounded-size set $\mathcal{S}$ of possible values); 2) runs the parametric reduction R_i; and 3) returns its result.

In such a setting, and for any event E, we will have, for every possible value j of the index

$$\Pr[\mathsf{R} : E] = |\mathcal{S}| \cdot \Pr[\mathsf{R} : E \wedge i = j]$$

The plug-and-pray library formalizes the argument generically, allowing arbitrary instantiations or the reduction, guess and event, and further allowing the value of the index to be derived from the state of the experiment, instead of being externally quantified.

A.2.2 Hybrids

The second result we heavily rely on in our ML-KEM proof is a generic hybrid argument, which captures generically union bound-style results in indistinguishability steps.

Consider two oracles $\mathcal{O}_l$ and $\mathcal{O}_r$ of the same type, and a distinguisher D trying to distinguish them in at most q queries. Our generic hybrid proves (entirely generically) that, for some distinguisher D_1 that makes at most 1 query to its oracle, we have

$$|\Pr[\mathsf{D}(\mathcal{O}_l)] - \Pr[\mathsf{D}(\mathcal{O}_r)]| \leq q \cdot |\Pr[\mathsf{D}_1(\mathcal{O}_l)] - \Pr[\mathsf{D}(\mathcal{O}_r)]|$$

The hybrid library formalizes the argument generically, allowing arbitrary instantiations of the oracle and distinguisher types, and also supporting the result where the oracle being replaced shares state with oracles that do not change from one side to the other.

A.2.3 PROM

The final generic result we use formally captures the cryptographic argument that "a value is random and independent of the adversary's view." This was initially produced as part of the formalization of indifferentiability from a random oracle for SHA3 [17], and is therefore framed as an indistinguishability result between two different implementations of a programmable random oracle interface. This result shows that no context (or unbounded adversary) can distinguish between the *eager* and *lazy* versions of the oracles presented in Figure A.4, where the latter omits line 1 of the sample algorithm. These oracles allow for globally initializing or resetting (init), or locally observing associated

Oracle init():	Oracle get(x):	Oracle set(x, r):	Oracle sample(x):
1: $m \leftarrow \varnothing$	1: **If**$(x \notin m)$ $m[x] \leftarrow\!\!\$\, R$	1: $m[x] \leftarrow r$	1: **If**$(x \notin m)$ $m[x] \leftarrow\!\!\$\, R$
2: **return** ()	2: **return** $m[x]$	2: **return** ()	2: **return** ()

Figure A.4: PROM oracles, in the lazy version the dashed box is removed

values (get), programming the random oracle (set), or eagerly (resp. lazily) sampling values associated with given entries (sample). In essence, the library establishes generically that sampling unused values can be deferred until they (or some behavior that depends on them) first become visible to the adversary or context.

The result itself is quite general, and hides away some of the complex program logics involved in proving it, allowing less expert users to more easily formalize complex proofs. However, its application in our setting, where we need to identify a specific query which will be sampled eagerly, is somewhat complex: we cannot simply sample the entire random function eagerly over an infinite domain, and neither can we predict the values of the queries the adversary will make. Most of the applications we make of this result in the ML-KEM proof first involve an indirection mapping query values to their index (essentially, and up to repeat queries, the first query would have index 1, the second query would have index 2, and so on), and eagerly sampling a map from integers (between 1 and the maximum number of queries). This essentially allows to partially sample the outputs of the function and later assign them to particular inputs as the adversary makes queries.

B

SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange

B.1 Proofs for Section 5.3 (Preliminaries)

Lemma 1 (Linear Independence of Vectors [203, Lem. 1]). Let $\mathcal{R}_q \coloneqq \mathbb{Z}_q[X]/(X^d+1)$ and let $\eta(f(X))$ denote the set of irreducible proper divisors of the polynomial $f(X) = X^d + 1 \in \mathbb{Z}_q$. For linearly independent vectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k$ where $\boldsymbol{u}_i \in \mathcal{R}_q^N$ and $0 \le k \le N-1$, the number of vectors $\boldsymbol{v} \in \mathcal{R}_q^N$ such that $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}\}$ is linearly independent is at most

$$\sum_{\alpha \in \eta(f)} q^{k \deg(f)} q^{(N-k)\deg(\alpha)}.$$

sketch. We estimate the maximum number of vectors $\boldsymbol{v} \in \mathcal{R}_q^N$ for which there exists a nonzero tuple of coefficients $\alpha_1, \ldots, \alpha_k, \alpha \in \mathcal{R}_q^{k+1}$ such that $\alpha_1 \boldsymbol{u}_1 + \cdots + \alpha_k \boldsymbol{u}_k = \alpha \boldsymbol{v}$. Consider the set $V_\alpha \coloneqq \{\boldsymbol{v} \in \mathcal{R}_q^N : \alpha \boldsymbol{v} \in \mathcal{L}(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k)\}$ for $\alpha \in \mathcal{R}_q$, where $\mathcal{L}(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k)$ denotes the linear hull of vectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k$. Note that $V_\alpha = V_{\alpha\beta}$ for any invertible $\beta \in \mathcal{R}_q$. Therefore the set of all different values of α may be split into equivalence relation $\alpha' \sim \alpha'' \iff (\exists$ an invertible $\beta : \alpha' = \beta\alpha'')$. Finally, one can show that

$$\begin{aligned} |V_\alpha| &\le |\mathbb{Z}_q[X]/(f/\alpha)|^k \cdot (q^{\deg \alpha})^N \\ &= q^{k \deg(f/\alpha)} q^{N \deg(\alpha)} \\ &= q^{k \deg f} q^{(N-k)\deg \alpha}. \end{aligned}$$

■

Qubits, n-qubit States and Measurement. A qubit $|x\rangle \coloneqq \alpha_0 |0\rangle + \alpha_1 |1\rangle$ is a unit vector in some Hilbert space $\mathcal{H}$. When $\alpha_0 \neq 1$ and $\alpha_1 \neq 1$, we say that $|x\rangle$ is in *superposition*. An n-bit quantum register $|x\rangle \coloneqq \sum_{i=0}^{2^n-1} \alpha_i |i\rangle$ is a unit vector in $\mathcal{H}^{\otimes n} \cong \mathbb{C}^{2^n}$, that is $\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$ for $\alpha_i \in \mathbb{C}$. We call the

set $\{|0\rangle, |1\rangle, \ldots, |2^n - 1\rangle\}$ the *computational basis* and say that $|x\rangle$ is *entangled* when $|x\rangle$ cannot be written as the tensor product of single qubits. Unless otherwise stated, measurements are done in the computational basis. After measuring a quantum register $|x\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle$ in the computational basis, the state *collapses* and $|x\rangle = \pm |i\rangle$ with probability $|\alpha_i|^2$.

Quantum Algorithms. A quantum algorithm A is a sequence of unitary operations U_i, where unitary operations are defined to map unit vectors to unit vectors, whilst preserving the normalisation constraint of quantum registers. A quantum oracle algorithm A^{O} is defined analogously, and can additionally query the oracle O before (or after) executing a unitary U_i. As quantum computations need to be reversible, we model an oracle $\mathsf{O} : X \to Y$ by a unitary U_{O} that maps $|x\rangle |y\rangle \mapsto |x\rangle |y \oplus \mathsf{O}(x)\rangle$. For an oracle O, we write $|\mathsf{O}\rangle$ to denote that an algorithm has quantum-access to U_{O}.

B.2 Proofs for Section 5.4 (Definitions)

Game HKR-CKS-l$_{\mathsf{NIKE},par}$

$b \xleftarrow{\$} \{0,1\}$
$b' \leftarrow \mathsf{A}^{|\mathsf{H}\rangle,\texttt{RegHonUsr}(\cdot),\texttt{TestQue}(\cdot,\cdot)}$
return $[\![b = b']\!]$

Figure B.1: Game defining **HKR-CKS-l**$_{\mathsf{NIKE},par}$ for a non-interactive key exchange NIKE relative to a random oracle H with adversary A. `RegHonUsr` and `TestQue` are defined in Figure B.2.

B.2.1 Non-Interactive Zero-Knowledge Proofs

Zero-Knowledge proofs [107] allow a verifier to convince a prover of the validity of a statement without revealing anything beyond that. In the random oracle model [42] zero-knowledge proofs can be made non-interactive [61] by applying the Fiat-Shamir transformation [96].

Definition 1 (Zero-Knowledge Proof of Knowledge). A zero-knowledge proof of knowledge ZKPoK for an NP language $\mathcal{L}$ [51] is defined as a tuple $\mathsf{ZKPoK} \coloneqq (\mathsf{ZK.Prv}, \mathsf{ZK.Ver})$ of the following oracle algorithms.

$\pi \xleftarrow{\$} \mathsf{ZK.Prv}^{\mathsf{H}}(x, w)$**:** Given a statement x and a witness w, the probabilistic prover algorithm $\mathsf{ZK.Prv}$ returns a proof π.

[51] The language $\mathcal{L}$ is defined as the set of all yes-instances of the relation R, i.e. $\mathcal{L} = \{x : \exists\, w \text{ s.t. } R(x, w) = 1\}$.

Game $\mathbf{CKS}_{\mathsf{NIKE},par}$ / $\mathbf{CKS\text{-}l}_{\mathsf{NIKE},par}$

$b \xleftarrow{\$} \{0,1\}$
$\mathcal{D} := \bot$
$\mathcal{K} := \bot$
$b' \leftarrow \mathsf{A}^{|\mathsf{H}),\ \mathtt{RegHonUsr}(\cdot),\ \mathtt{RegCorUsr}(\cdot,\cdot),\ \mathtt{RevCorQue}(\cdot,\cdot),\ \mathtt{TestQue}(\cdot,\cdot)}$
return $[\![b = b']\!]$

Oracle $\mathtt{RegHonUsr}(\mathtt{ID} \notin \mathcal{IDS})$ // Only twice in **CKS-l**

if $(corrupt, \mathtt{ID}, \bot, \cdot) \in \mathcal{D}$:
  return $\bot$
$(sk, pk) \xleftarrow{\$} \mathsf{Gen}(\mathtt{ID})$
$\mathcal{D} \cup \{(honest, \mathtt{ID}, sk, pk)\}$
return pk

Oracle $\mathtt{RegCorUsr}(\mathtt{ID} \notin \mathcal{IDS}, pk)$ // $Q_{\mathtt{RCU}}$ queries

if $(corrupt, \mathtt{ID}, \bot, \cdot) \in \mathcal{D}$:
  $(corrupt, \mathtt{ID}, \bot, \cdot) := (corrupt, \mathtt{ID}, \bot, pk)$
else :
  $\mathcal{D} \cup \{(corrupt, \mathtt{ID}, \bot, pk)\}$

Oracle $\mathtt{RevCorQue}(\mathtt{ID}_1 \notin \mathcal{IDS}, \mathtt{ID}_2 \notin \mathcal{IDS})$ // $Q_{\mathtt{RCQ}}$ queries

if $(honest, \mathtt{ID}_1, \cdot, \cdot) \in \mathcal{D} \wedge (corrupt, \mathtt{ID}_2, \cdot, \cdot) \in \mathcal{D}$:
  return $\mathsf{SdK}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, sk_1)$
elseif $(corrupt, \mathtt{ID}_1, \cdot, \cdot) \in \mathcal{D} \wedge (honest, \mathtt{ID}_2, \cdot, \cdot) \in \mathcal{D}$:
  return $\mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2)$
return $\bot$

Oracle $\mathtt{TestQue}(\mathtt{ID}_1 \notin \mathcal{IDS}, \mathtt{ID}_2 \notin \mathcal{IDS})$ // Only 1 in **CKS-l**

if $\mathtt{ID}_1 = \mathtt{ID}_2$:
  return $\bot$
if $(honest, \mathtt{ID}_1, \cdot, \cdot) \in \mathcal{D} \wedge (honest, \mathtt{ID}_2, \cdot, \cdot) \in \mathcal{D}$:
  if $b = 0$:
    $k := \mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2)$
  if $b = 1$:
    if $(\mathtt{ID}_1, \mathtt{ID}_2, k) \in \mathcal{K} \vee (\mathtt{ID}_2, \mathtt{ID}_1, k) \in \mathcal{K}$:
      return k
    $k \xleftarrow{\$} \mathcal{SKS}$
    $\mathcal{K} \cup \{(\mathtt{ID}_1, \mathtt{ID}_2, k)\}$
  return k
return $\bot$

Figure B.2: Game defining $\mathbf{CKS}_{\mathsf{NIKE},par}$ (and $\mathbf{CKS\text{-}l}_{\mathsf{NIKE},par}$) for a non-interactive key exchange NIKE with adversary A.

$1/0 \leftarrow \mathsf{ZK.Ver}^{\mathsf{H}}(x, \pi)$**:** Given a statement x and a proof π, the deterministic verifier algorithm returns either 1 for accept or 0 for reject.

Similar to the work of [200] we assume a distribution RODist on functions, modelling the distribution of our random oracle. That is, given a random

oracle $\mathsf{H} : \{0,1\}^* \to \{0,1\}^n$, RODist would be the uniform distribution on $\{0,1\}^* \to \{0,1\}^n$.

ZKPoK Security Notions. Besides *completeness*, which captures that valid proofs are accepted by the verifier, a zero-knowledge proof of knowledge should fulfill two additional properties; *soundness* ensures a cheating prover cannot convince the verifier of a false proof, and *zero-knowledge* conveys that the verifier learns nothing from its interaction with the prover beyond the fact that he knows a valid witness to the proof. We make this more precise with the following definitions and note that we require the strong notion of *simulation soundness with a straight-line extractor* [200], sometimes referred to as "online extractability" in the literature.

Definition 2 (Completeness). *Completeness* for a zero-knowledge proof of knowledge ZKPoK of an NP language $\mathcal{L}$ is defined via the game $\mathbf{CMPLT}_{\mathsf{ZKPoK}}$ depicted in Figure B.3. For an adversary A, we define A's advantage in $\mathbf{CMPLT}_{\mathsf{ZKPoK}}$ as

$$\mathsf{Adv}^{\mathbf{CMPLT}}_{\mathsf{ZKPoK}}(\mathsf{A}) \coloneqq \Pr[\mathbf{CMPLT}^{\mathsf{A}}_{\mathsf{ZKPoK}} \Rightarrow 1],$$

and say that ZKPoK is $(\epsilon, Q_{\mathsf{H}})$-$\mathbf{CMPLT}$ if for all quantum-polynomial-time adversaries A, making at most Q_{H} queries (possibly in superposition) to the random oracle H, we have $\mathsf{Adv}^{\mathbf{CMPLT}}_{\mathsf{ZKPoK}}(\mathsf{A}) \leq \epsilon(Q_{\mathsf{H}})$.

Game $\mathbf{CMPLT}_{\mathsf{ZKPoK}}$

$\mathsf{H} \xleftarrow{\$} \mathsf{RODist}$
$(x, w) \xleftarrow{\$} \mathsf{A}^{|\mathsf{H}\rangle}$
$\pi \xleftarrow{\$} \mathsf{ZK.Prv}^{\mathsf{H}}(x, w)$
return $[\![\mathsf{ZK.Ver}^{\mathsf{H}}(x, \pi) = 0 \wedge R(x, w) = 1]\!]$

Figure B.3: Game defining $\mathbf{CMPLT}_{\mathsf{ZKPoK}}$ for a zero-knowledge proof of knowledge ZKPoK with adversary A.

For the following notions we additionally require a simulator $\mathsf{ZK.Sim} \coloneqq (\mathsf{ZK.Sim}_1, \mathsf{ZK.Sim}_2)$ that is split into two classical algorithms $\mathsf{ZK.Sim}_1$ and $\mathsf{ZK.Sim}_2$, where:

$\mathsf{H} \xleftarrow{\$} \mathsf{ZK.Sim}_1$**:** The probabilistic simulator algorithm $\mathsf{ZK.Sim}_1$ returns a circuit H which represents the initial simulated random oracle.

$\pi \xleftarrow{\$} \mathsf{ZK.Sim}_2(x)$**:** Given a statement x the stateful simulator algorithm $\mathsf{ZK.Sim}_2$ returns a proof π. Additionally, $\mathsf{ZK.Sim}_2$ is given access to the description of H and may replace it with a different description (i.e. it can program the random oracle).

Definition 3 (Zero-Knowledge [107]). *Zero-knowledge* for a zero-knowledge proof of knowledge ZKPoK of an NP language $\mathcal{L}$ is defined via the game $\mathbf{ZK}^b_{\mathsf{ZKPoK}}$, depicted in Figure B.4, where $\mathbf{ZK}^b_{\mathsf{ZKPoK}}$ is parametrised by a bit b. For an adversary A, we define A's advantage in $\mathbf{ZK}^b_{\mathsf{ZKPoK}}$ as

$$\mathsf{Adv}^{\mathbf{ZK}}_{\mathsf{ZKPoK}}(\mathsf{A}) \coloneqq \left|\Pr[\mathbf{ZK}^{0,\mathsf{A}}_{\mathsf{ZKPoK}} \Rightarrow 1] - \Pr[\mathbf{ZK}^{1,\mathsf{A}}_{\mathsf{ZKPoK}} \Rightarrow 1]\right|,$$

and say that ZKPoK is (ϕ, Q_{H})-$\mathbf{ZK}$, if there exists a PPT simulator $\mathsf{ZK.Sim} \coloneqq (\mathsf{ZK.Sim}_1, \mathsf{ZK.Sim}_2)$, such that for all quantum-polynomial-time adversaries A, making at most Q_{H} queries (possibly in superposition) to the random oracle H, we have $\mathsf{Adv}^{\mathbf{ZK}}_{\mathsf{ZKPoK}}(\mathsf{A}) \leq \phi(Q_{\mathsf{H}})$.

Game $\mathbf{ZK}^0_{\mathsf{ZKPoK}}$	**Game** $\mathbf{ZK}^1_{\mathsf{ZKPoK}}$	**Procedure** $\mathsf{ZK.Sim}'_2(x, w)$
01 $\mathsf{H} \xleftarrow{\$} \mathsf{RODist}$	04 $\mathsf{H} \xleftarrow{\$} \mathsf{ZK.Sim}_1$	07 **if** $R(x, w) = 0$:
02 $b' \xleftarrow{\$} \mathsf{A}^{\lvert\mathsf{H}\rangle, \mathsf{ZK.Prv}(\cdot,\cdot)}$	05 $b' \xleftarrow{\$} \mathsf{A}^{\lvert\mathsf{H}\rangle, \mathsf{ZK.Sim}'_2(\cdot,\cdot)}$	08 **return** $\perp$
03 **return** $[\![b' = 0]\!]$	06 **return** $[\![b' = 1]\!]$	09 **else** :
		10 **return** $\mathsf{ZK.Sim}_2(x)$

Figure B.4: Games defining $\mathbf{ZK}^b_{\mathsf{ZKPoK}}$ for a zero-knowledge proof of knowledge ZKPoK with adversary A and simulator $\mathsf{ZK.Sim} \coloneqq (\mathsf{ZK.Sim}_1, \mathsf{ZK.Sim}_2)$. The purpose of $\mathsf{ZK.Sim}'_2(\cdot,\cdot)$ is merely to serve as an interface for the adversary who expects a prover taking two arguments x and w.

Definition 4 (Simulation-Sound Online-Extractability [200]). *Simulation-sound online-extractability* [52] for a zero-knowledge proof of knowledge ZKPoK of an NP language $\mathcal{L}$ is defined via the game $\mathbf{SSND}_{\mathsf{ZKPoK}}$, depicted in Figure B.5. For an adversary A, we define A's advantage in $\mathbf{SSND}_{\mathsf{ZKPoK}}$ as

$$\mathsf{Adv}^{\mathbf{SSND}}_{\mathsf{ZKPoK}}(\mathsf{A}) \coloneqq \Pr[\mathbf{SSND}^{\mathsf{A}}_{\mathsf{ZKPoK}} \Rightarrow 1],$$

and say that ZKPoK is (ψ, Q_{H})-$\mathbf{SSND}$ relative to a simulator $\mathsf{ZK.Sim} \coloneqq (\mathsf{ZK.Sim}_1, \mathsf{ZK.Sim}_2)$, if there exists a PPT extractor $\mathsf{ZK.Ext}$ such that for all quantum-polynomial-time adversaries A, making at most Q_{H} queries to the random oracle H, we have $\mathsf{Adv}^{\mathbf{SSND}}_{\mathsf{ZKPoK}}(\mathsf{A}) \leq \psi(Q_{\mathsf{H}})$.

B.3 Proofs for Section 5.5 (Construction)

Combining Theorem 3 with the passive version of Theorem 1 immediately implies the following Corollary.

Corollary 1. *For any* PPT *adversary* A *against* $\mathsf{NIKE} \coloneqq (\mathsf{Stp}, \mathsf{Gen}, \mathsf{SdK})$*, depicted in Figure 5.3, making an arbitrary number of queries (possibly in*

[52] Online-extractability is sometimes referred to as straight line extractability in the literature.

Game $\mathbf{SSND}_{\mathsf{ZKPoK}}$

$\mathsf{H} \xleftarrow{\$} \mathsf{ZK.Sim}_1$
$(x, \pi) \xleftarrow{\$} \mathsf{A}^{|\mathsf{H}\rangle, \mathsf{ZK.Sim}_2(\cdot)}$
$w \xleftarrow{\$} \mathsf{ZK.Ext}(\mathsf{H}, x, \pi)$
return $[\![\mathsf{ZK.Ver}^{\mathsf{H}}(x, \pi) = 1 \wedge R(x, w) = 0 \wedge (x, \pi) \notin \tilde{\boldsymbol{\pi}}]\!]$

Figure B.5: Games defining $\mathbf{SSND}_{\mathsf{ZKPoK}}$ for a zero-knowledge proof of knowledge ZKPoK with adversary A, simulator $\mathsf{ZK.Sim} \coloneqq (\mathsf{ZK.Sim}_1, \mathsf{ZK.Sim}_2)$ and extractor ZK.Ext. Here, $\tilde{\boldsymbol{\pi}}$ denotes the set of all proofs returned by $\mathsf{ZK.Sim}_2(\cdot)$ (together with the corresponding statements).

superposition) to H*,* $Q_{\texttt{RHU}}$ *(classical) queries to the* `RegHonUsr` *oracle and one (classical) query to the* `TestQue` *oracle, there exist* PPT *adversaries* B_1, B_2 *such that*

$$\mathsf{Adv}^{\mathbf{HKR\text{-}CKS}}_{\mathsf{NIKE},par}(\mathsf{A}) \leq \frac{Q^2_{\texttt{RHU}} \cdot Q_{\texttt{TQ}}}{2} \cdot \Big(\mathsf{Adv}^{\mathbf{M\text{-}LWE}}_{q,N,N,\chi}(\mathsf{B}_1) + \mathsf{Adv}^{\mathbf{M\text{-}LWE}}_{q,N,N+1,\chi}(\mathsf{B}_2) + \frac{4\beta d}{q}\Big).$$

B.3.1 Proof of Theorem 4

Note that it suffices to prove **CKS-l** and then apply Corollary 1 to obtain the result for **CKS**. Let A be an adversary against NIKE in the **CKS-l** game. Consider the sequence of games in Figure B.6, where $Q_{\texttt{TQ}} = 1$ and $Q_{\texttt{RHU}} = 2$ denote the number of queries to `TestQue` and `RegHonUsr`, respectively.

Game G_0 This is the original $\mathbf{CKS\text{-}l}_{\mathsf{NIKE},par}$ game, so by definition

$$\Pr\left[\mathsf{G}^{\mathsf{A}}_0 \Rightarrow 1\right] = \Pr\left[\mathbf{CKS\text{-}l}^{\mathsf{A}}_{\mathsf{NIKE},par} \Rightarrow 1\right].$$

Game G_1 In this game we modify the `RegCorUsr` oracle so that the secret key $\widetilde{sk'_{\texttt{ID}}}$ is extracted from the proof π of a public key pk on Line 16, and stored with the identity `ID`. This requires the strong notion of simulation-sound online-extractability, from Definition 4. If the extraction fails, then by default $\widetilde{sk'_{\texttt{ID}}} = \perp$. I.e. the secret key is not stored, as in the original game. Therefore,

$$\left|\Pr\left[\mathsf{G}^{\mathsf{A}}_0 \Rightarrow 1\right] - \Pr\left[\mathsf{G}^{\mathsf{A}}_1 \Rightarrow 1\right]\right| \leq Q_{\texttt{RCU}} \cdot \mathsf{Adv}^{\mathbf{SSND}}_{\mathsf{ZKPoK}}(\mathsf{B}_1).$$

Games G_0, G_1, G_2, G_3, G_4, G_5

Proof.
$b \xleftarrow{\$} \{0,1\}$
$\mathcal{D} := \bot$
$\mathcal{K} := \bot$
$b' \leftarrow \mathsf{A}^{|\mathsf{H}\rangle, \mathtt{RegHonUsr}(\cdot), \mathtt{RegCorUsr}(\cdot,\cdot), \mathtt{RevCorQue}(\cdot,\cdot), \mathtt{TestQue}(\cdot,\cdot)}$
return $[\![b = b']\!]$

Oracle $\mathtt{RegHonUsr}(\mathtt{ID} \in \mathcal{IDS})$

// Twice

if $(corrupt, \mathtt{ID}, \bot, \cdot) \in \mathcal{D}$:
  return $\bot$
$(sk'_{\mathtt{ID}}, pk'_{\mathtt{ID}}) \xleftarrow{\$} \mathsf{Gen}(\mathtt{ID})$
$\pi \xleftarrow{\$} \mathsf{ZK.Prv}(pk'_{\mathtt{ID}}, sk'_{\mathtt{ID}})$
$\pi \xleftarrow{\$} \mathsf{ZK.Sim}_2(pk'_{\mathtt{ID}})$ // G_5
$sk_{\mathtt{ID}} := sk'_{\mathtt{ID}}$
$pk_{\mathtt{ID}} := (pk'_{\mathtt{ID}}, \pi)$
$\mathcal{D} \cup \{(honest, \mathtt{ID}, sk_{\mathtt{ID}}, pk_{\mathtt{ID}})\}$
return $pk_{\mathtt{ID}}$

Oracle $\mathtt{RegCorUsr}(\mathtt{ID} \in \mathcal{IDS}, pk)$

// $Q_{\mathtt{RCU}}$ queries

parse $pk \to (pk'_{\mathtt{ID}}, \pi)$
$\widetilde{sk'_{\mathtt{ID}}} \xleftarrow{\$} \mathsf{ZK.Ext}(\mathsf{H}, pk'_{\mathtt{ID}}, \pi)$ // G_1
if $(corrupt, \mathtt{ID}, \cdot, \cdot) \in \mathcal{D}$:
  $(corrupt, \mathtt{ID}, \bot, \cdot) := (corrupt, \mathtt{ID}, \bot, pk)$
  $(corrupt, \mathtt{ID}, \cdot, \cdot) := (corrupt, \mathtt{ID}, \widetilde{sk'_{\mathtt{ID}}}, pk)$ // G_1
else :
  $\mathcal{D} \cup \{(corrupt, \mathtt{ID}, \bot, pk)\}$
  $\mathcal{D} \cup \left\{\left(corrupt, \mathtt{ID}, \widetilde{sk'_{\mathtt{ID}}}, pk\right)\right\}$ // G_1

Oracle $\mathtt{RevCorQue}(\mathtt{ID}_1 \in \mathcal{IDS}, \mathtt{ID}_2 \in \mathcal{IDS})$ // $Q_{\mathtt{RCQ}}$ queries

if $(honest, \mathtt{ID}_1, \cdot, \cdot) \in \mathcal{D} \wedge (corrupt, \mathtt{ID}_2, \cdot, \cdot) \in \mathcal{D}$:
  if $\mathsf{SdK}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, sk_1) \neq \mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2)$: // G_2-G_3
    abort // G_2-G_3
  return $\mathsf{SdK}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, sk_1)$
  return $\mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, \widetilde{sk'_2})$ // G_3
elseif $(corrupt, \mathtt{ID}_1, \cdot, \cdot) \in \mathcal{D} \wedge (honest, \mathtt{ID}_2, \cdot, \cdot) \in \mathcal{D}$:
  if $\mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2) \neq \mathsf{SdK}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, sk_1)$: // G_2-G_3
    abort // G_2-G_3
  return $\mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2)$
  return $\mathsf{SdK}(\mathtt{ID}_2, pk_2, \mathtt{ID}_1, \widetilde{sk'_1})$ // G_3
return $\bot$

Oracle $\mathtt{TestQue}(\mathtt{ID}_1 \in \mathcal{IDS}, \mathtt{ID}_2 \in \mathcal{IDS})$ // Once

if $\mathtt{ID}_1 = \mathtt{ID}_2$:
  return $\bot$
if $(honest, \mathtt{ID}_1, \cdot, \cdot) \in \mathcal{D} \wedge (honest, \mathtt{ID}_2, \cdot, \cdot) \in \mathcal{D}$:
  if $b = 0$:
    $k := \mathsf{SdK}(\mathtt{ID}_1, pk_1, \mathtt{ID}_2, sk_2)$
  if $b = 1$:
    if $(\mathtt{ID}_1, \mathtt{ID}_2, k) \in \mathcal{K} \vee (\mathtt{ID}_2, \mathtt{ID}_1, k) \in \mathcal{K}$:
      return k
    $k \xleftarrow{\$} \mathcal{SKS}$
    $\mathcal{K} \cup \{(\mathtt{ID}_1, \mathtt{ID}_2, k)\}$
  return k
return $\bot$

Figure B.6: Games G_1-G_5, for the proof of **CKS-l** of NIKE in Figure 5.5.

Game G_2 In this game we introduce a new condition for aborting: If at any point in the simulation the adversary asks a query the `RevCorQue` oracles on two public keys that cause a key mismatch, then abort the simulation.

Note that this condition is efficiently testable, as the game knows all the secret keys. It is important to note that the strong notion of **SM-COR** quantifies over all IDs, meaning we can bound the probability of this event happening with a reduction to the semi-malicious correctness property, Definition 4, of NIKE' by

$$\left|\Pr\left[\mathsf{G}_1^{\mathsf{A}} \Rightarrow 1\right] - \Pr\left[\mathsf{G}_2^{\mathsf{A}} \Rightarrow 1\right]\right| \leq \mathsf{Adv}^{\mathbf{SM\text{-}COR}}_{\mathsf{NIKE}',par'}(\mathsf{B}_2).$$

Game G_3 In this game we modify the `RevCorQue` oracle on Line 27 and Line 32 to use the secret key that was extracted when the corresponding public key was registered as a corrupt key.

Since this is just a conceptual change and the derived key is always the same for both secret keys, we get

$$\left|\Pr\left[\mathsf{G}_2^{\mathsf{A}} \Rightarrow 1\right] - \Pr\left[\mathsf{G}_3^{\mathsf{A}} \Rightarrow 1\right]\right| = 0.$$

Game G_4 In this game we undo the modification from G_2 and get

$$\left|\Pr\left[\mathsf{G}_3^{\mathsf{A}} \Rightarrow 1\right] - \Pr\left[\mathsf{G}_4^{\mathsf{A}} \Rightarrow 1\right]\right| \leq \mathsf{Adv}^{\mathbf{SM\text{-}COR}}_{\mathsf{NIKE}',par'}(\mathsf{B}_2).$$

Game G_5 In this game we modify the `RegHonUsr` oracle and replace the zero-knowledge proof of knowledge on Line 10 with a simulated proof. Recall that we are considering **CKS-l**, meaning only 2 calls are made to `RegHonUsr`. By the zero-knowledge property, Definition 3, we get

$$\left|\Pr\left[\mathsf{G}_4^{\mathsf{A}} \Rightarrow 1\right] - \Pr\left[\mathsf{G}_5^{\mathsf{A}} \Rightarrow 1\right]\right| \leq 2 \cdot \mathsf{Adv}^{\mathbf{ZK}}_{\mathsf{ZKPoK}}(\mathsf{B}_3).$$

Adversary $\mathsf{C}^{|\mathsf{H}\rangle,\mathtt{RegHonUsr}(\cdot),\mathtt{TestQue}(\cdot,\cdot)}$

01 $b' \leftarrow \mathsf{A}^{|\mathsf{H}\rangle,\ \mathtt{SimRegHonUsr},\ \mathtt{SimRegCorUsr},\ \mathtt{SimRevCorQue},\ \mathtt{SimTestQue}}$
02 **return** b'

Oracle $\mathtt{SimRegCorUsr}(\mathtt{ID} \in \mathcal{IDS}, pk)$ // Same as G_5

Oracle $\mathtt{SimRegHonUsr}(\mathtt{ID} \in \mathcal{IDS})$

03 $pk' \leftarrow \mathtt{RegHonUsr}(\mathtt{ID})$
04 $\pi \xleftarrow{\$} \mathsf{ZK.Sim}_2(pk')$
05 $pk \coloneqq (pk', \pi)$
06 **return** pk

Oracle $\mathtt{SimRevCorQue}(\mathtt{ID}_1, \mathtt{ID}_2)$ // Same as G_5 **Oracle** $\mathtt{SimTestQue}(\mathtt{ID}_1, \mathtt{ID}_2)$

07 **return** $\mathtt{TestQue}(\mathtt{ID}_1, \mathtt{ID}_2)$

Figure B.7: Adversary C for bounding G_5.

At this point we no longer need the secret keys to simulate and can show a reduction. Next we make the following Claim.

Claim. There exists a PPT adversary C such that

$$\Pr\left[\mathsf{G}_5^{\mathsf{A}} \Rightarrow 1\right] = \mathsf{Adv}_{\mathsf{NIKE}}^{\textbf{HKR-CKS-l}}(\mathsf{C}).$$

Proof. The adversary C, depicted in Figure B.7, against **HKR-CKS-l** is constructed as follows. Any queries that A in G_5 makes to `SimRegHonUsr` and `SimTestQue`, C forwards to the `RegHonUsr` and `TestQue` oracles in the **HKR-CKS-l** game, except that C adds a simulated proof to the public key returned by the `RegHonUsr` oracle. On the other hand, the queries of A to the `RegCorUsr` and `RevCorQue` oracles are simulated by C as in G_5, since no secret key is required to compute their output. The same guess bit b' returned by A is also returned by C. It is easy to see that the reduction perfectly simulates the view of the adversary and therefore any advantage of A directly carries over to the C. ■

Collecting all the probabilities and applying Theorem 1 yields the bounds from the Theorem statement, concluding the proof. ■

C

Jazzline: Composable CryptoLine functional-correctness proofs for Jasmin programs

C.1 Background on CryptoLine

We give a summary of the main CryptoLine features and refer the interested reader to [199, 99] for the full details.

Types represent unsigned and signed bitwords of fixed width. For width w, they are inhabitted by integers i in the range $0 \leq i < 2^w$ for uint and $-2^{w-1} \leq i < 2^{w-1}$ for sint. A CryptoLine program consists of variable declarations followed by instructions. A typical instruction retrieves values from sources/atoms and stores them in destinations/variables. An atom is either a variable or a constant. Instructions include operations that are pervasive across architectures. Other instructions, such as splitting and joining bitwords to obtain bitwords of different sizes, can be used to model additional processor operations. assert and assume instructions can also occur at any point in a program, both expressed as the conjunction of two types of predicates: range predicates and algebraic predicates. We consider only straight-line CryptoLine programs.

The CryptoLine type system is, for the most part, straightforward. Many operations, namely arithmetic instructions, are closed over the type (i.e. word size and signedness). Unsigned operations have the expected typing, and only for signed operations there are some important nuances that aim to guarantee consistency with the interpretation over the integers of the behavior of some operations.

The semantics of CryptoLine are standard when it comes to statements/instructions. The interpretation of the value of a variable depends on its type. The semantics of instructions are defined by interpreting the inputs as integers, performing a computation over the integers, and then injecting the result back into one or more bitwords of the correct types. If the integer result cannot be represented in the output variables, this yields an explicit error.

The semantics of an assert statement is to leave the current state unchanged if the associated predicate evaluates to true in this state, and an error otherwise. The semantics of an assume statement is to leave the current state unchanged if the associated predicate evaluates to true in this state, and it is undefined behavior otherwise (for simplicity, one can think of assuming false as an instruction that blocks the execution of the program). Note that in the CryptoLine semantics, the evaluation of algebraic predicates is carried out over the integers. For range predicates, the predicate validity is checked directly over bit-vectors.

CryptoLine is geared towards automatic verification using two families of tools: SMT solvers supporting the bit vector theory SMT-LIB2, and Computer Algebra Systems (CAS). The CryptoLine tool can convert a partial correctness Hoare triple proof goal for a CryptoLine program into a set of sufficient proof obligations. Note that partial correctness here means that the post-condition holds whenever the evaluation of the program reached a final state. Because the program is straight-line and verified to be safe, the only possible cause for not reaching a final state is blocking due to the attempt to evaluate an assume statement which turns out to be false. Proof obligations associated with range predicates are discharged using SMT solvers. Obligations associated with algebraic predicates are discharged using CAS. The safety of a CryptoLine program is proved using a dedicated verification condition generator that checks for the property that the CryptoLine program does not terminate in an error state.

Summary

The security of cryptographic systems relying on classical problems such as factoring large integers or computing discrete logarithm problems, and their respective implementations, has historically been addressed through the use of pen-and-paper proofs and software testing techniques. However, recent advancements in quantum computers have raised concerns regarding the security of these cryptographic systems. The ability to efficiently solve the problems these systems rely on, with approaches such as Shor's algorithm, has motivated research into quantum-resistant cryptography, also called post-quantum cryptography. Concurrent with this transition, and driven by the aspiration to attain more robust guarantees regarding the security of these systems than those provided by pen-and-paper proofs and conventional software testing techniques, computer-aided cryptography offers a means to address this challenge through the utilization of formal methods. The present thesis is an examination of the application of techniques from computer-aided cryptography to these new cryptographic primitives, as well an advancement of the status of post-quantum primitives for key exchange.

In this line of research, in Chapters 3 and 4 we provide the first end-to-end formally verified implementation of the ML-KEM standard. This implementation utilizes the Jasmin framework and EasyCrypt proof assistant, representing a new advancement in the field of formally verified cryptography.

Secondly, in Chapter 5 we undertake an exploratory investigation into the feasibility of lattice-based NIKEs. This includes the construction of an efficient passively secure NIKE and its subsequent conversion to an actively secure one through the utilization of a generic compiler. The efficacy of this approach is illustrated through an optimized implementation of the passively secure construction in Rust.

Finally, in Chapter 6 we propose a novel technique to automate formal proofs of functional correctness for algebraic code by combining EasyCrypt, Jasmin, and CryptoLine. The applicability of this approach is demonstrated through the presentation of the first verified implementation of X-Wing, as well as through the presentation of proofs of correctness for certain optimized components of the Jasmin implementation of Kyber described in Chapter 3.

Samenvatting

De veiligheid van cryptografische systemen die gebaseerd zijn op klassieke problemen zoals het ontbinden van grote getallen in factoren of het berekenen van discrete logaritmeproblemen, en de respectieve implementaties daarvan, werd in het verleden aangepakt door middel van bewijzen op papier en softwaretesttechnieken. Recente ontwikkelingen op het gebied van kwantumcomputers hebben echter vragen doen rijzen over de veiligheid van deze cryptografische systemen. De mogelijkheid om de problemen waarop deze systemen zijn gebaseerd efficiënt op te lossen, met benaderingen zoals het algoritme van Shor, heeft aanleiding gegeven tot onderzoek naar kwantumbestendige cryptografie, ook wel post-kwantumcryptografie genoemd. Gelijktijdig met deze transitie, en gedreven door de ambitie om robuustere garanties te verkrijgen met betrekking tot de veiligheid van deze systemen dan de garanties die worden geboden door bewijzen op papier en conventionele softwaretesttechnieken, biedt computerondersteunde cryptografie een manier om deze uitdaging aan te gaan door gebruik te maken van formele methoden. Dit proefschrift is een onderzoek naar de toepassing van technieken uit de computerondersteunde cryptografie op deze nieuwe cryptografische primitieven, evenals een verbetering van de status van post-kwantumprimitieven voor sleuteluitwisseling.

In deze onderzoekslijn bieden we in hoofdstukken 3 en 4 de eerste end-to-end formeel geverifieerde implementatie van de ML-KEM-standaard. Deze implementatie maakt gebruik van het Jasmin-framework en de EasyCrypt-bewijsassistent, wat een nieuwe vooruitgang betekent op het gebied van formeel geverifieerde cryptografie.

Ten tweede voeren we in hoofdstuk 5 een verkennend onderzoek uit naar de haalbaarheid van op roosters gebaseerde NIKE's. Dit omvat de constructie van een efficiënte passief veilige NIKE en de daaropvolgende conversie naar een actief veilige NIKE door middel van een generieke compiler. De doeltreffendheid van deze aanpak wordt geïllustreerd door een geoptimaliseerde implementatie van de passief veilige constructie in Rust.

Ten slotte stellen we in hoofdstuk 6 een nieuwe techniek voor om formele bewijzen van functionele correctheid voor algebraïsche code te automatiseren

door EasyCrypt, Jasmin en CryptoLine te combineren. De toepasbaarheid van deze aanpak wordt aangetoond door de presentatie van de eerste geverifieerde implementatie van X-Wing, evenals door de presentatie van bewijzen van correctheid voor bepaalde geoptimaliseerde componenten van de Jasmin-implementatie van Kyber, zoals beschreven in hoofdstuk 3.

About the Author

Miguel was born in Évora, Portugal on June 23, 1997. He studied computer engineering at the University of Minho, where he completed his B.Sc. in 2018. He started a master's degree the same year, also at University of Minho, and concluded it in 2020. His M.Sc. thesis on "TrustZone based Attestation in Secure Runtime Verification for Embedded Systems" was completed under the supervision of José Carlos Bacelar Almeida. Before starting his Ph.D., he worked at Goldman Sachs as a Cybersecurity Security Analyst. In 2021 he started his Ph.D. studies on high-assurance post-quantum cryptography at the Max Planck Institute for Security and Privacy, under the supervision of Gilles Barthe and Peter Schwabe. This thesis contains the result of his work from February 2021 to May 2025.

List of Publications

- José Bacelar Almeida and Manuel Barbosa and Gilles Barthe and Benjamin Grégoire and Vincent Laporte and Jean-Christophe Léchenet and Tiago Oliveira and Hugo Pacheco and Miguel Quaresma and Peter Schwabe and Antoine Séré and Pierre-Yves Strub. "Formally verifying Kyber – Episode IV: Implementation correctness". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (June 2023), pp. 164–193. DOI: 10.46586/tches.v2023.i3.164-193

- José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe and Pierre-Yves Strub. "Formally Verifying Kyber - Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt". In: *Advances in Cryptology – CRYPTO 2024, Part II*. ed. by Leonid Reyzin and Douglas Stebila. Vol. 14921. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2024, pp. 384–421. DOI: 10.1007/978-3-031-68379-4_12

- Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta and Peter Schwabe. "SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange". In: *USENIX Security 2024: 33rd USENIX Security Symposium*. Ed. by Davide Balzarotti and Wenyuan Xu. Philadelphia, PA, USA: USENIX Association, Aug. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/gajland

- José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Lionel Blatter, Gustavo Delerue, João Diogo Duarte, Benjamin Gregoire, Tiago Oliveira, Miguel Quaresma, Pierre-Yves Strub, Ming-Hsien Tsai, Bow-Yaw Wang and Bo-Yin Yang. "Jazzline: Composable CryptoLine Functional Correctness Proofs for Jasmin Programs". In: *ACM CCS 2025: 32nd Conference on Computer and Communications Security*. ACM Press, Nov. 2025, pp. 1409–1423. DOI: 10.1145/3719027.3744814