Post-Quantum Cryptography in Resource-Constrained Environments

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

dinsdag 9 juni 2026
om 16:30 uur precies

door

Ruben Anthony Gonzalez

Promotoren:

prof. dr. P. Schwabe

Manuscriptcommissie:

prof. dr. L. Batina	(voorzitter)
dr. D.F. Aranha	*Aarhus Universitet, Denemarken*
prof. dr. Y. Yarom	*Ruhr-Universität Bochum, Duitsland*
prof. dr. D.F. Oswald	*Durham University, Verenigd Koninkrijk*
dr. D. Stebila	*University of Waterloo, Canada*

Contents

Contents iii

1 Introduction 1
1.1 Post-Quantum Cryptography 2
1.2 Resource-Constrained Environments 3
1.3 Contributions 3

2 Preliminaries 7
2.1 Cryptography 7
2.1.1 Definitions and Notation 7
2.1.2 Symmetric Primitives 8
2.1.3 Asymmetric Primitives 11
2.1.4 Protocols 13
2.2 Post-Quantum Cryptography 14
2.2.1 NIST Competition 15
2.2.2 Algorithm Families 16
2.3 Resource-Constrained Environments 18
2.3.1 Embedded Systems 18
2.3.2 Key Metrics in Embedded Systems 19
2.3.3 Cortex-M Processor Family 21
2.3.4 Embedded Software 21
2.3.5 Evaluation Platforms and Data Collection 22

I Implementing Primitives 25

3 Verifying Post-Quantum Signatures in 8 kB of RAM 27
3.1 Introduction 28
3.2 Background 29
3.2.1 Feature Activation 29
3.2.2 Alternative Implementation 31
3.3 Analyzed Post-Quantum Signature Schemes 31
3.3.1 Hash-Based Schemes 32
3.3.2 Multivariate-Based Schemes 34
3.3.3 Lattice-Based Schemes 35
3.4 Implementation 37
3.4.1 Streaming Interface 37

3.4.2 Public Key Verification 37
3.4.3 Implementation Details 37
3.5 Results 43

4 Formally Verified Zeroization 45
4.1 Introduction 47
4.2 Failure Modes 50
4.2.1 Perform No Zeroization 51
4.2.2 Zeroization Falling Prey to Compiler Optimizations . . . 51
4.2.3 Zeroization in API Functions Only 52
4.2.4 Zeroization on Source Level 53
4.3 Possible Solutions 55
4.3.1 Caller-Side Zeroization 55
4.3.2 Callee-Side Zeroization 55
4.4 A principled solution in the compiler 56
4.4.1 Background on Jasmin 56
4.4.2 Design Choices 58
4.4.3 Implementation Overview 59
4.4.4 Register and Flag Zeroization 62
4.4.5 Correctness and Security 62
4.4.6 Combining Leakage Models 63
4.5 Benchmarks and Validation 65
4.6 Conclusion and Future Work 68

II Securing Protocols 71

5 KEMTLS vs. Post-Quantum TLS in Resource-Constrained Devices 73
5.1 Introduction 75
5.2 Background 76
5.2.1 Post-Quantum TLS 77
5.3 Experimental Setup 79
5.3.1 Implementation 80
5.4 Results 82
5.4.1 Storage and Memory Consumption 82
5.4.2 Handshake Times 83
5.5 Discussion 89
5.6 Conclusion and Future Work 89

6 Post-Quantum FIDO with Hash-Based Signatures 91
6.1 Introduction 93
6.1.1 Related Work 93
6.2 Background 95
6.2.1 FIDO-Based Authentication 95
6.2.2 Post-Quantum Cryptography 97

6.2.3 SPHINCS^{+} . . . 98
6.3 Implementation . . . 101
6.3.1 Requirements . . . 102
6.3.2 Adjusting SPHINCS^{+} . . . 103
6.4 Results . . . 105
6.4.1 SPHINCS^{+} Instantiations . . . 105
6.4.2 Hash Functions . . . 105
6.4.3 Adjusted SPHINCS^{+} Benchmarks . . . 107
6.4.4 FIDO Benchmarks . . . 110
6.4.5 Discussion and Future Work . . . 111
6.5 Conclusion . . . 111

III Supplementary Works 113

7 Security Analysis of the Olvid Secure Messenger 115
7.1 Introduction . . . 116
7.2 Preliminaries . . . 119
7.2.1 Security models for Key Exchange . . . 119
7.2.2 Security models for CKA . . . 120
7.2.3 The Tamarin prover . . . 120
7.2.4 Cache side channel attacks . . . 121
7.3 The Olvid Encrypted Messenger . . . 121
7.3.1 Threat Model . . . 123
7.3.2 The cryptographic core of Olvid . . . 123
7.4 Formal analysis using Tamarin . . . 127
7.4.1 Formally modeling Olvid . . . 127
7.4.2 Positive findings . . . 131
7.4.3 Negative findings . . . 134
7.5 Source-code analysis and implementation security . . . 136
7.5.1 Positive findings . . . 136
7.5.2 Negative findings . . . 136
7.6 Conclusion and Future Work . . . 138
7.6.1 Avenues for future work . . . 141

8 Conclusion and Outlook 143

Acronyms 145

Bibliography 147

Research Data Management 177

Summary / Samenvatting 179

Chapter 1

Introduction

> *"Cryptography is the art of transforming a complex information security problem into a complex key management problem."*
>
> *— Someone in the hallway at Real World Crypto*

Cryptography is the study of algorithms that protect information from an intelligent adversary. As "protect" is a rather ambiguous term, computer scientists have divided it into three main goals: **Confidentiality** assures that only intended recipients can read a given information, **integrity** guarantees that no information was altered and **authenticity** makes sure that information stems from the inferred sender and not an impostor. These goals are a necessity for secure digital communication — from bank transactions, car keys and confidential messaging to remote access into critical infrastructures. They can be met using cryptographic techniques.

On a high level, cryptography works by securing information with a key. This can be done with a single secret key that is shared amongst all communication partners. This style of operation is referred to as **symmetric** cryptography and is inherently limited in its scalability. Distributing symmetric keys has been tedious and error-prone work in the past, often relying on confidential code books or other difficult means of transportation. To combat this limitation, a new approach to cryptography was proposed: **Asymmetric** cryptography refers to algorithms that do not require a secret key to be shared for secure communication. Instead, every communication partner uses **keypairs**, consisting of a private and public key. The public key can be shared with the whole world and still be used to establish secure communication with the possessor of its associated private key. In digital communication systems, asymmetric cryptography is commonly used in conjunction with long-lasting keypairs to establish a secure channel. Within this channel, a symmetric secret key is exchanged to protect individual messages.

Using symmetric algorithms for protecting the actual information payload

within an asymmetrically established channel is done for performance reasons. Asymmetric cryptography tends to be more expensive than its symmetric counter part in terms of computational resources, such as memory, computational time and transmission size. Asymmetric algorithms are expensive mainly because they are based on involved mathematical constructions. Namely, the security of virtually all asymmetric algorithms in use today is based on the hardness of two mathematical problems. These are the so-called **prime factorization** and the **discrete logarithm** problems. Conversely, if these two problems would be efficiently solvable for an adversary, our digital communication would lose confidentiality, integrity and authenticity. In fact, both problems are quite related, as they are both instances of the superjacent abelian hidden subgroup problem in the additive group of integers. Maybe surprisingly, this means that an efficient solution to this single problem would shatter our digital civilization by rendering asymmetric cryptography useless.

And so it appears that asymmetric cryptography, a corner stone of our interconnected world, suffers from two major deficiencies. First and foremost, it lacks diversity by relying on the supposed hardness of a single mathematical problem. Secondly, it is expensive in terms of computational resources. Tackling both deficiencies is a major goal of contemporary cryptography research.

1.1 Post-Quantum Cryptography

Shor further fanned the flames of doubt with his 1994 publication "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer" [Sho94]. With his work, Shor expounded that a large-enough quantum computer could be leveraged to efficiently solve the problem that underpins widely-used asymmetric cryptography. This means that, if a large quantum computer is engineered, the asymmetric cryptography we rely on would be entirely broken. Continued advances in quantum computing make this an intolerable risk.

Given the looming threat of quantum computing, the term ***Post-Quantum Cryptography*** (PQC) was coined to describe a new set of algorithms built on assumptions that shall also withstand an adversary in possession of large quantum computers [Ber09]. In 2016, the government of the United States of America acknowledged the problems of asymmetric cryptography. Fearing threats to national security, the U.S. National Institute of Standards and Technology (NIST) issued a call to the international cryptography community, asking for new algorithms [Nat16]. After the cryptography community answered the call by proposing a variety of algorithms, a multi-year selection process involving discussions about the feasibility, efficiency and security of these proposals was set on track. First tangible results emerged in 2024 with the standardization of new asymmetric algorithms [Nat24a] [Nat24b] [Nat24c]. These algorithms serve the same purpose as previous asymmetric algorithms but do not rely on the same mathematical assumption as their predecessors.

1.2 Resource-Constrained Environments

While these algorithms add security, they also pose new engineering challenges. Tuning these complex algorithms to decrease transmission size, memory consumption, storage space and runtime is a main topic of study in the contemporary field of cryptography engineering. Optimizing these algorithms for embedded devices is especially challenging. Unlike general-purpose computers, embedded devices are produced for a specific purpose in a specific environment, have tight hardware constraints and usually run dedicated software optimized for resource efficiency and reliability.

Their environments typically impose further restrictions regarding resources such as available power or communication bandwidth. Examples of embedded devices running in resource-constrained environments can be found in most areas of our lives. An insulin pump is as much of an embedded device as an automotive airbag controller. Not seldom, lives depend on the authenticity and integrity of these devices and the information they store, send, and receive. In fact, connecting embedded devices to large computer networks has become so common that observers devised the phrase "Internet of Things". The number of network-connected embedded devices even outweighs that of general-purpose computers such as laptops or smartphones by far. Securing embedded devices and the information they carry and exchange with the resource-intense tools of asymmetric cryptography therefore poses a difficult but paramount challenge.

1.3 Contributions

This thesis aims to contribute to the field of applied cryptography by bringing post-quantum cryptography to real-world, resource-constrained environments. The primary focus of this work is on quantum-secure authentication. Excellent previous work has contributed numerous advances in optimizing specific parts of isolated, embedded algorithm implementations [KRSS]. This work aims to build up on that research and paint a broader picture. The word "environment" in its title is to be taken literally. In the following chapters, post-quantum cryptography algorithms are not investigated in an isolated fashion on a given embedded platform. Instead, they are deployed into embedded environments that fulfil a security-critical role in a technical context. All of these environments are constrained in their resources to deliver insights into a variety of embedded use cases. I hope the following additions to the literature will prove helpful for academia and industry.

The content of this thesis is based on peer-reviewed papers that were written interdependently or in collaboration with coauthors. These papers will be introduced in the following sections, together with a statement about my original contribution.

Part I: Post-Quantum Implementation

The first part of the thesis zooms in on the implementation of (post-quantum) primitives in specific, resource-constrained contexts. It is mainly based on two peer-reviewed papers. The use case of feature activation in cars using a strong head unit and a weak trusted platform module (TPM) with very few memory is investigated in Chapter 3. It is based on the paper:

> Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. Verifying Post-Quantum signatures in 8 kB of RAM. In *Post-Quantum Cryptography – PQCrypto 2021*, LNCS, pages 215–233. Springer, 2021. `https://eprint.iacr.org/2021/662`

In it, various post-quantum signature algorithm implementations are aligned to meet the tight time and memory constrains. To accomplish this, the signatures are streamed into a simulated TPM which verifies them for correctness. Implementation of post-quantum primitives were done in equal parts by Mathias Kannwischer and me. The benchmarking framework used for this unusual purpose was designed and implemented by me. Paper writing was distributed between all authors, with me contributing mainly to the implementation and results sections.

Part I concludes with Chapter 4, based on a joint paper about clearing sensitive data, such as key material, from memory in cryptography implementations. In the paper's introduction, the problem of leaking critical cryptographic values via stack spillage is introduced. Then case studies are presented, showing that this can be a problem in real world cryptography libraries and investigate how those libraries try to mitigate the threat. A solution using the Jasmin programming language is then presented and evaluated for multiple primitives.

> Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, and Peter Schwabe. High-assurance zeroization. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):375–397, 2024. `https://eprint.iacr.org/2023/1713`

I contributed the research and documentation on how real world libraries are affected by this problem and what mitigations they have in place. During this research I also identified vulnerabilities in the OpenSSL and libsodium cryptography libraries that are documented in the paper and have been responsibly disclosed to the OpenSSL and libsodium developers. Further, I provided an example application and proof of concept exploit showing that this vulnerability can be exploited in real software systems. For the written paper I contributed the sections about failure modes and possible solutions in collaboration with Peter Schwabe.

Part II: Post-Quantum Protocols for Embedded Systems

While the previous part was concerned with implementation of post-quantum primitives in a specific setting, this part of the thesis deals with existing, standardized protocols utilizing asymmetric cryptography in an embedded setting. Its content is mainly based on two papers giving insight into how post-quantum authentication could be achieved in those protocols and how that would affect performance. In collaboration with Thom Wiggers, the following paper was devised.

> Ruben Gonzalez and Thom Wiggers. KEMTLS vs. post-quantum TLS: Performance on embedded systems. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 99–117. Springer, 2022. `https://eprint.iacr.org/2022/1712`

As the title suggests, it contains performance comparisons between post-quantum TLS and KEMTLS-based server authentication in resource constrained environments. These environments are limited in computational resources and feature an array of different transmission technologies offering various communication bandwidths and latencies. Thom Wiggers, who is the principle author behind KEMTLS [SSW20], kindly contributed Section 5.2 introducing TLS and KEMTLS as part of the background. He also supplied latex/tikz code for the visualization and tables of benchmarks. I have contributed the implementation of TLS and KEMTLS with post-quantum authentication for embedded environments, the reproducible benchmarking setup, a benchmarking integration into the ZephyrOS real-time operating system, benchmarking results, analysis and all remaining paper sections. Both TLS and KEMTLS implementations are based on the WolfSSL embedded library and feature an altered version of the PQM4 [KRSS] library offering WolfSSL-compatible APIs.

The second paper, contained within Chapter 6, deals with post-quantum user authentication. Within this single author paper, the SPHINCS+ stateless hash-based signature algorithm [ABB+22] is instantiated with a variety of parameters to show that it can be a surprisingly competitive choice when tailored to a specific, resource constrained environment. The experiments were conducted on real hardware and used the FIDO2/CTAP2 user authentication mechanisms.

> Ruben Gonzalez. Stateless hash-based signatures for post-quantum security keys. In *International Conference on Applied Cryptography and Network Security*, volume 15654 of *LNCS*. Springer, 2025. `https://eprint.iacr.org/2025/298`

Idea, implementation, benchmarks, analysis and all sections are my original work. However, it is noteworthy that the implementation and benchmarking setup is based on Google's OpenSK experimental FIDO2 implementation. Additionally, I received valuable feedback from my supervisor Peter Schwabe.

Part III: Supplementary Works

This final part offers a view beyond the embedded realm. In it, the cryptographic security of the Olvid messenger [Olv25a] is analyzed. The paper focuses on Olvid's cryptographic core, specifically its protocols for authenticated key exchange. Using the symbolic prover Tamarin, various security goals of Olvid are formally verified. Additionally, a time-based side channel vulnerability in Olvid's implementation is described and some design choices are called into question.

> Noemi Terzo, Cas Cremers, Ruben Gonzalez, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. Security analysis of the olvid secure messenger. *unpublished*

For this work, I assisted in the reverse engineering of the Olvid messenger app and its communication protocols. I further contributed to the paper's sections about Olvid's inner workings, its source code analysis and certain implementation-specific shortcomings.

Chapter 2

Preliminaries

This chapter presents concepts, definitions and notation necessary to understand the subsequent chapters. We start with Section 2.1, introducing cryptographic terminology and definitions, before introducing background on Post-Quantum Cryptography in Section 2.2. Section 2.3 then gives background on embedded systems and experimental design.

2.1 Cryptography

On first sight, just like mathematicians, cryptographers seem to enjoy confusing people with complex notation. But, of course, this supposedly complex notation is needed to eliminate ambiguity and enable efficient communication. For the same purpose, this thesis makes use of some standard definitions and notation. The following section introduces the terminology, definitions and notation needed to understand the following chapters.

As this thesis is mainly concerned with performance and implementation security, formal security models and proofs are largely omitted in the preliminaries. The reader is encouraged to consult the literature, for example [MF21], for further background on these topics.

2.1.1 Definitions and Notation

Cryptography tries to accomplish security goals using algorithms that employ keys. We therefore begin by specifying keyed algorithms and our definition of security. When using a keyed algorithm Alg, we denote the employed key k as Alg_k. In algorithm definitions, we use $\perp$ as failure symbol, meaning the computation did not conclude to a valid output.

This work is primarily focused on performance results of post-quantum cryptography algorithms and not their underlying security proofs. So for us it suffices to define *security* via the number of operations an adversary would need to break a cryptographic system using a particular attack. We assume

this "particular attack" to be the best known attack against the system, unless stated otherwise. Definition 2.1 introduces *bit security* as used throughout the thesis.

Definition 2.1 (Bit Security). *A cryptographic system has a bit security of* n *against a given adversary* $\mathcal{A}$ *mounting a particular attack, if* $\mathcal{A}$ *requires at least* 2^n *operations to gain a relevant chance to break at least one security property.*

The mathematical notation in this thesis follows established conventions. Throughout this thesis, variable assignment is denoted as $variable \leftarrow value$ and variable equality with $var_1 = var_2$. To assign a value by drawing randomly from a given set, we denote $variable \leftarrow_{\$} set$. Cardinality of a set is denoted as $|set|$. The same notation is sometimes used to express the length of a cryptographic object in bytes, as in $|sig| = 32$ specifying the length of a given signature as 32 bytes. $a \| b \| c$ denotes the concatenation of byte strings a, b and c with Hello‖World = HelloWorld. We define $\{1, ..., n\}$ as the set of integers from 1 to n and specifically $\{0,1\}^l$ as a bit string of length l and $\{0,1\}^*$ as a bit string of arbitrary length. The term $\mathbb{Z}_q$ represents the additive group of integers modulo q, $\mathbb{Z}_q^*$ the group of integers modulo q under multiplication.

2.1.2 Symmetric Primitives

While the focus of this thesis is primarily on post-quantum – and hence asymmetric – cryptography, symmetric primitives are necessary building blocks for the discussed protocols. This section briefly introduces the required background.

Symmetric primitives are characterized by the fact that all communication parties use the same inputs to their computations. Typically, this means a secret key, that is shared amongst communication parties, is used to encrypt **and** decrypt, to sign **and** verify. For this reason, symmetric cryptography is sometimes referred to as *secret-key cryptography.* Commonly, symmetric algorithms are more performant than their asymmetric counter parts and are therefore trusted with protection of payload data, whereas asymmetric algorithms are used in involved protocols to establish symmetric keys between the communicating parties.

Hash Function

An idealized hash function maps an arbitrary-length input to a fixed-length output. A cryptographic hash function additionally offers at least three security guarantees. Informally, for a polynomial-time, probabilistic adversary $\mathcal{A}$ it must be hard to break:

Collision resistance by finding arbitrary inputs m, m' that lead to the same output,

pre-image resistance by finding an input m for a given output and

second pre-image resistance by finding an input m' that produces the same output as given input m.

Definition 2.2 (Cryptographic Hash Function). *A hash function*

$$H(m) \rightarrow \{0,1\}^l, \text{ for } m \in \{0,1\}^*$$

that offers collision resistance, pre-image resistance and second pre-image resistance.

From this we arrive at Definition 2.2 for a cryptographic hash function. Cryptographic hash functions are used for various duties in cryptographic protocols, but are also a necessary building block for most asymmetric primitives, such as digital signatures (see Section 2.1.3).

Key-Derivation Function

A key-derivation function (KDF) is used to cast key material into a key. KDFs are mainly used to align key material to the correct length and uniformity (e.g. to be used as a symmetric key) or to derive multiple distinct keys from the same key material. The key material used may stem from an asymmetric algorithm as described in Section 2.1.3. Throughout this work, the Hash-based KDF (HKDF) construction as defined in [Kra10b], based on [Kra10a], is used. HKDF is an extract-then-expand-style KDF as described in Definition 2.3.

Definition 2.3 (Extract-then-Expand Key Derivation Function). *An extract-then-expand KDF implements the algorithms:*

$\mathsf{KDF.Extract}(\mathsf{IKM}, \mathsf{salt}) \rightarrow \mathsf{PRK}$ *an extract algorithm where the input keying material* IKM *is used with an optional, non-secret* salt *value to derive a pseudo-random key* PRK*.*

$\mathsf{KDF.Expand}(\mathsf{PRK}, \mathsf{info}, l) \rightarrow \mathsf{OKM}$ *an expand algorithm that delivers output keying material of length l based on a* PRK *(usually the output of extract) and some non-confidential context data,* info*.*

It is important to note here, that the PRK is not inherently “stronger” than the IKM, as a KDF does not add entropy. However, given sufficient entropy in IKM, KDF.Extract outputs a value PRK that is hard to distinguish from uniform. In this work, $\mathsf{KDF.Extract}(\mathsf{IKM}, \mathsf{salt})$ may be abbreviated as $\mathsf{KDF}(\mathsf{IKM})$ (with empty salt) or $\mathsf{KDF}(\mathsf{IKM}, \mathsf{salt})$ for readability in diagrams.

Message Authentication Code

Message authentication codes (MACs) preserve the authenticity of messages. For the remainder of this work, we use the term "*authenticity*" as the guarantee of integrity (the message was not altered) and authentication (the message stems from the inferred sender). MACs are symmetric primitives as the sender and receiver of a given message use the same secret key to compute an *authentication tag*.

Definition 2.4 (Message Authenciation Code)**.** *A message authentication code consists of three probabilistic polynomial-time algorithms:*

$\mathsf{MAC.KeyGen}(1^l) \rightarrow k$ *a key generation algorithm that, on input the security parameter l, outputs a secret key k.*

$\mathsf{MAC.Mac}_k(m) \rightarrow t$ *a MAC generation algorithm that takes as input a secret key k and a message $m \in \{0,1\}^*$, and outputs a tag t.*

$\mathsf{MAC.Verify}_k(m,t) \rightarrow \{0,1\}$ *a verification algorithm that takes as input the secret key k, a message m, and a tag t, and outputs* 1 *if t is a valid tag for m under k, and* 0 *otherwise.*

Upon receiving a message with its associated authentication tag, the receiver recomputes the tag using the message and shared secret key. He then proceeds to compare the received authentication tag with the recomputed value. Given they are identical, the message is considered authentic. For Definition 2.5 we model this verification as a distinct algorithm. MACs are frequently used in combination with symmetric encryption to guarantee confidentiality and authenticity of messages.

Authenticated Encryption

Authenticated Encryption with Associated Data (AEAD) algorithms offer a single primitive to assure confidentiality and authenticity. Usually they employ a form of symmetric encryption and a MAC construction to accomplish this. Our definition given below is based on [Rog02].

Definition 2.5 (Authenticated Encryption with Associated-Data)**.** *A authenticated encryption with associated data scheme consists of three probabilistic polynomial-time algorithms:*

$\mathsf{AEAD.KeyGen}(1^l) \rightarrow k$ *a key generation algorithm that takes a security parameter l and outputs a secret key k.*

$\mathsf{AEAD.Enc}_k(n,a,m) \rightarrow c$ *an encryption algorithm that takes as input a secret key k, a unique nonce $n \in \{0,1\}^*$, associated data $a \in \{0,1\}^*$, plaintext $m \in \{0,1\}^*$ and outputs a ciphertext $c \in \{0,1\}^*$ with the associated data being authenticated, but not encrypted.*

$\mathsf{AEAD.Dec}_k(n, a, c) \rightarrow m$ **or** $\perp$ *a decryption algorithm that takes as input the secret key k, nonce n, associated data a, and ciphertext c, and outputs either the plaintext message m or a failure symbol $\perp$ in case of MAC verification failure.*

The associated data in AEAD schemes is meant to transport necessary metadata that does not require confidentiality, e.g. certain protocol headers. However, in this work we do not use associated data at all. For our purpose, the employed AEAD schemes are used as authenticated encryption (AE). Within the text, nonces are sometimes omitted for brevity. $\mathsf{AEAD}_k(m)$ or $\mathsf{AE}_k(m)$ in following diagrams therefore refers to $\mathsf{AEAD.Enc}_k(n, a, m)$ with empty associated data and random nonce.

2.1.3 Asymmetric Primitives

Unlike symmetric primitives that rely on a single shared key, asymmetric cryptography uses a pair of linked keys: typically one public and one private. Crucially, revealing a public key does not affect the security of the system or the confidentiality of the private key. This section introduces the algorithm definitions used throughout the thesis.

Digital Signature Scheme

Asymmetric digital signature schemes are used to ensure authenticity, without relying on a shared secret key. Definition 2.6 specifies such schemes on a high level.

Definition 2.6 (Digital Signature Scheme). *A digital signature scheme consists of three algorithms:*

$\mathsf{Sig.KeyGen}(1^l) \rightarrow (\mathsf{sk}, \mathsf{pk})$ *a key generation algorithm that takes a security parameter l and outputs a key pair with private key sk and public key pk.*

$\mathsf{Sig.Sign}(\mathsf{sk}, m) \rightarrow \mathsf{S}$ *a signature algorithm that takes a private key sk and message $m \in \{0,1\}^*$ and outputs a signature S.*

$\mathsf{Sig.Verify}(\mathsf{pk}, m, \mathsf{S}) \rightarrow \{0,1\}$ *a verification algorithm that takes public key pk, message m, signature S and outputs 1 only if S is a valid signature of m under pk.*

In its post-quantum call, NIST actually asked for submissions with a different definition [Nat16]. NIST followed the more misuse-resistant approach of defining a Sign function returning a single value containing signature and message and an Open function that consumes this value and only returns the message given the signature is valid. This has the advantage that inexperienced or negligent developers cannot access the message directly in case signature verification fails. However, we use the definition above because existing protocols (see Part II) and code bases for our conducted experiments depend on it.

Changing APIs from a Sign/Verify to Sign/Open paradigm, and vice versa, is usually a triviality anyway and does not affect performance results. For brevity in diagrams, $\mathsf{Sig.Sign}(\mathsf{sk}, m)$ may be written as $\mathsf{Sign}_{\mathsf{sk}}(m)$.

Diffie-Hellman Key Exchange

Current versions of security protocols, such as TLS, mostly rely on the Diffie-Hellman (DH) key exchange [DH76] for securely exchanging a symmetric key. DH can, in principle, be conducted in any finite cyclic group in which it is infeasible to compute logarithms. Common choices for these groups are $\mathbb{Z}_q^*$ with prime q or prime-order subgroups of points on elliptic curves. The latter have no known sub-exponential time attacks and therefore offer smaller keys and faster computation. Therefore, using DH with elliptic curves is the de-facto standard for asymmetric key exchange as of today. Cryptography utilizing elliptic curves is commonly referred to as Elliptic Curve Cryptography (ECC). Definition 2.7 formally introduces DH. While it is customary to use additive notation for elliptic curves, the definition uses the multiplicative notation from the original publication. This is done for consistency with the remaining definitions.

Definition 2.7 (Diffie-Hellman Key Exchange). *Given the parties have agreed on security parameters: $(\mathbb{G}, n, g)$ with finite cyclic group $\mathbb{G}$ of order n and generator g, the Diffie-Hellman key exchange consists of two algorithms:*

$\mathsf{DH.KeyGen}() \rightarrow (\mathsf{sk}, \mathsf{pk})$ *a key generation algorithm that, based on the security parameters $(\mathbb{G}, n, g)$, selects a secret exponent $x \leftarrow\!\!\$\, \mathbb{Z}_n^*$, computes $X = g^x \in \mathbb{G}$, and outputs the key pair $(\mathsf{sk} = x, \mathsf{pk} = X)$.*

$\mathsf{DH.Exchange}(\mathsf{sk}, \mathsf{pk}_p) \rightarrow s$ *a key exchange algorithm that, on input of a private key $sk = x$ and the other party's public key $pk_p = X_p = g^{x_p} \in \mathbb{G}$, outputs the shared secret $s = X_p^x = g^{x_p x} = X^{x_p} \in \mathbb{G}$.*

It is easy to see that a passive adversary observing transmitted public keys cannot compute the shared secret s, unless he is able to compute the discrete logarithm $\log_g(pk) = sk$ for one of them. While this seems to be a hard problem for classical computers, Shor's algorithm [Sho94] enables quantum computers to do exactly that. The susceptibility of DH to quantum attacks is one of the strongest driving forces behind the adoption of post-quantum cryptography.

Commonly, DH keypairs are ephemeral. They are not persisted to long-term storage and created fresh for every new iteration of the algorithm. This serves the purpose of *forward secrecy* [DvW92], as an adversary that compromises a communication partner will not be able to compute previously exchanged secret keys (and hence compromise confidentiality) based on observed key exchanges.

Key-Encapsulation Mechanism

As relying on traditional DH has become dangerous, a different, more abstract definition for key exchange primitives is needed. Traditionally, public

key encryption (PKE), was considered an alternative. PKE schemes encrypt a symmetric key via a public key. The possessor of the corresponding private key can then decrypt and use the symmetric key. While the post-quantum key exchange schemes employed in this thesis are based on PKE constructions, we use their established Key-Encapsulation Mechanism (KEM) form. PKEs can be transformed into KEMs, partially gaining stronger security notions, using generic transforms [FO99, HHK17].

Definition 2.8 (Key-Encapsulation Mechanism)**.** *A Key-Encapsulation Mechanism consists of three algorithms*

$\mathsf{KEM.KeyGen}(1^l) \rightarrow (\mathsf{sk}, \mathsf{pk})$ *a keypair generation algorithm that takes a security parameter l and outputs a keypair with private key* sk *and public key* pk.

$\mathsf{KEM.Encapsulate}(\mathsf{pk}) \rightarrow (s, c)$ *an encapsulation algorithm that takes a public key* pk *as input and outputs a shared secret s and a ciphertext c.*

$\mathsf{KEM.Decapsulate}(\mathsf{sk}, c) \rightarrow s$ *a decapsulation algorithm that takes a private key* sk *and ciphertext c and outputs the shared secret s.*

Such KEMs can be used as drop-in replacements for DH in most places. To do so, the communication partners exchange a public key and a ciphertext instead of two public keys. The initiator can then decrypt the ciphertext, using his private key, to arrive at the shared secret.

2.1.4 Protocols

Protocols define a sequence of messages and computations to accomplish a goal. Within this thesis, protocols are described as message sequence charts. Figure 2.1 gives an example of such a chart with two communication parties. In it, Alice and Bob determine their future relationship based on a party invitation by Alice and a random response by Bob.

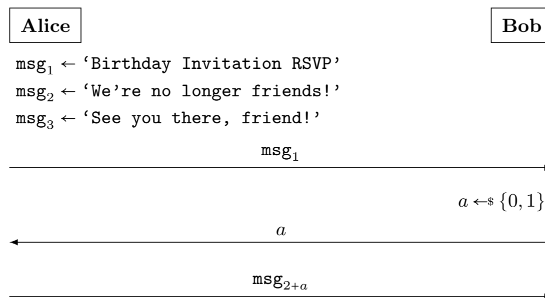


Figure 2.1: Example message sequence chart.

Security protocols usually combine cryptographic primitives to guarantee confidentiality and authenticity. As previously stated, common security protocols use symmetric cryptography, mostly AEAD schemes, to protect actual payload data. Before that can happen however, usually, a *handshake* is conducted within the protocol to establish symmetric key material. During the *handshake phase*, most protocols use asymmetric cryptography. This makes the handshake vital to security but also expensive. This is especially problematic in resource-constrained environments. Some resource-constrained environments, therefore use so-called Pre-Shared Key (PSK) solutions, where the symmetric key is exchanged out-of-band or is simply preinstalled on an embedded device, avoiding asymmetric cryptography altogether. Apart from scalability issues of distributing, revoking and managing symmetric keys per device, pure PSK solutions also do not offer forward secrecy. We therefore do not consider PSK solutions as viable options in this thesis.

Solely relying on DH or KEMs during the handshake phase is problematic, too. As public keys of these schemes are usually ephemeral, they are not known to the communication partner. An active adversary can therefore mount an impersonation attack, changing public keys to their own. This *man in the middle* can then read all exchanged data. Authenticated Key Exchange (AKE) [DvW92] protocols are used to avoid this kind of scenario. They combine establishment of a shared secret key with authentication of the communication partner. This is commonly done by either using long-term static public keys that are verified out-of-band, or by including digital signatures of a trust anchor into the handshake. TLS, for example, commonly uses the latter approach combined with an active handshake signature. Section 5.2.1 describes this further.

2.2 Post-Quantum Cryptography

Diversification of assumptions in asymmetric cryptography is the overarching goal of post-quantum cryptography (PQC) as a field of research. Symmetric cryptography is much less affected by known quantum attacks. The main concern for symmetric cryptography in general, Grover's quantum algorithm [Gro96], offers a quadratic speedup for generic search problems, slashing bit security (Definition 2.1) in half. However, this threat can be efficiently mitigated for symmetric cryptography by doubling the key length. As symmetric algorithms are typically very performant, this is feasible even in embedded settings.

PQC therefore aims to design algorithms with the same utility as pre-quantum asymmetric algorithms, but based on mathematical problems assumed to be hard for quantum computers as well. Since 2016, NIST has tried to identify and standardize the most promising post-quantum algorithms, with the help of the international cryptography community. In their call for PQC [Nat16], they asked for algorithms of two categories: (1) key establish-

ment using public key encryption or key encapsulation mechanisms and (2) digital signatures. Cryptographers from around the world answered the call and submitted over 80 proposals. The vast majority of these proposals can be categorized into five families, based on their underlying hardness assumption.

The following subsections give background on the NIST standardization process and its current status. Then, the five main PQC families will be briefly introduced with their advantages and disadvantages.

2.2.1 NIST Competition

In 2016, NIST announced its plans to replace the SP800-56 [Nat18, Nat19] and FIPS186-4 [Nat13] standards for key establishment and digital signatures based on integer factorization and the discrete logarithm. They were supposed to be succeeded by post-quantum algorithm standards of the same utility. To further diversify, NIST announced plans to standardize not one, but multiple schemes after a transparent multi-year selection process. The international cryptography community was asked to submit proposals until November 30th, 2017. As in previous standardization efforts, NIST asked for submissions with multiple parameter sets reflecting their security levels. Definitions of these security levels are detailed in Table 2.1 and linked to the hardness of breaking symmetric algorithms.

Table 2.1: NIST-defined security levels for the PQC standardization process.

Level	**Definition**
I	At least as hard to break as AES128 (exhaustive key search).
II	At least as hard to break as SHA256 (collision search).
III	At least as hard to break as AES192 (exhaustive key search).
IV	At least as hard to break as SHA384 (collision search).
V	At least as hard to break as AES256 (exhaustive key search).

Most of the 82 initially submitted algorithms specified parameter sets of security level I, III and V or exclusively III and V. In 2022, after three rounds of examination, NIST selected the first four algorithms to be standardized. The first standards, ML-KEM [Nat24a], ML-DSA [Nat24b], and SLH-DSA [Nat24c] followed in 2024. A fourth round concluded in 2025 with the selection of an additional key establishment algorithm called HQC [MAB+18]. Currently, NIST conducts an additional selection process for more post-quantum signature schemes [Nat22a]. As of now, there are five algorithms selected for standardization and fourteen in the second round of additional signature schemes. The work on this thesis began, when algorithms were already proposed, but none had been selected for standardization yet. Table 2.2 gives an overview of NIST's post-quantum selection process at the time of writing. The families mentioned in the table are introduced in the next section.

Table 2.2: Post-quantum algorithms standardized or selected for standardization and signature algorithms still considered by algorithm family.

Family	**Standardized / Selected**		**Still Considered**
	KEM	Signature	Signature (#)
Lattice	Kyber [SAB+22]	Dilithium [LDK+22] Falcon [PFH+22]	1
Code	HQC [MAB+18]	-	4
Isogeny	-	-	1
Multivariate	-	-	6
Symmetric	-	SPHINCS+ [HBD+22]	1

2.2.2 Algorithm Families

The bulk of proposed algorithms can be categorized into five families: lattice-based, code-based, isogeny-based, multivariate-based and symmetric-based. In the following subsections, these families are introduced briefly. In-depth introductions are presented in the corresponding chapters of the thesis.

Lattice-Based

Lattice-based schemes represent the majority of submissions for NISTs first PQC call. This family seems to be a dominating force, as three of the five algorithms selected for standardization are based on the hardness of lattice problems. These are the KEM Kyber [SAB+22] alongside signature schemes Dilithium [LDK+22] and Falcon [PFH+22]. Typically, lattice-based algorithms offer a good balance between speed and key sizes. Although their key sizes tend to be larger than that of their pre-quantum counter part, the computational performance is comparable. Depending on the CPU platform, they can even outperform pre-quantum algorithms in most metrics. Especially CPUs with vector instructions allow schemes like Kyber or Dilithium to shine. Unfortunately, embedded platforms do not have this privilege and other means are necessary to accelerate these algorithms. Detailed descriptions of lattice-based schemes relevant for this work can be found in Section 3.3.3.

Code-Based

McEliece coined this family of schemes based on error-correcting codes in 1978 [McE78]. Their hardness is based on the fact that decoding a random linear code is NP-hard. McEliece's original cryptosystem *hides* the decoding algorithm needed for a given encoded message. It is easy to see how that could lead to an encryption system. Code-based schemes are usually also performant on modern CPUs, but they tend to have larger keys and/or ciphertexts compared to lattice-based schemes. Managing such large keys on an embedded

device is usually challenging or even impossible (see e.g. Chapter 5). The HQC construction is code-based and was selected for standardization as a KEM.

Isogeny-Based

Isogeny-based crypto has been in the academic discourse only since 2006 [Cou06, RS06, CGL06]. It relies on graphs of isogenies between elliptic curves. Trust in this form of cryptography only grows slowly, due to its novelty and complexity. The break of the NIST round-4 candidate SIKE with a devastating attack [CD22] did not help in this regard. Isogeny-based schemes have comparatively small keys, ciphertexts and signatures [CLM+18, FKL+20]. However, their implementation is very complex and slower than their lattice and code-based counter parts. Due to implementation complexity, low performance and security concerns, isogeny-based cryptography does not play a significant role in the embedded sphere as of now.

Multivariate-Based

Multivariate signature schemes are based on the hardness of finding solutions to systems of equations in many variables over finite fields. The first such scheme was proposed by Matsumoto and Imai [MI88] in 1988 and promptly broken by Patarin [Pat95]. Submitted multi-variate schemes suffered a similar faith in the NIST selection process. The signature scheme Rainbow [DCK+19] was broken for all efficient parameters in round 3 [Beu22] and LUOV [BPSV19], GeMSS [CFM+20] and MQDSS [SCH+19] were eliminated from the selection because of successes in cryptanalysis [DDVY21, KZ20, TPD20]. Round 2 of the additional signature selection process contains submissions closer to the better understood construction of unbalanced oil and vinegar (UOV) [KPG99]. Multivariate schemes usually offer competitive performance and small signatures, but very large keys. This can be of utility in certain embedded scenarios, as further described in Section 3.3.2.

Symmetric-Based

Symmetric-based schemes employ symmetric primitives under the hood. Their security therefore relies on usually-well-understood assumptions about those underlying primitives. That is why they are commonly considered a conservative choice for post-quantum cryptography. Hash-based signature systems are part of this family and relevant for this work. Traditionally, hash-based constructions were *stateful*, meaning that a global state per key has to be kept track of [Lam79, BDH11]. Reusing a key's global state could lead to a compromise of the entire system. SPHINCS+ [HBD+22] on the other hand is a *stateless* hash-based system. It works by using a huge number of internal keys, so that picking a key at random leads to a negligible chance of reuse. SPHINCS+ was selected for standardization by NIST and is detailed in Section 6.2.3. Hash-based systems usually suffer from comparatively low performance and large

signatures, but benefit from very small keys. Still, due to their use of symmetric primitives, which can sometimes be accelerated in hardware, and their conservative security assumptions they can be of utility for embedded systems (see e.g. Chapter 3 and Chapter 6).

2.3 Resource-Constrained Environments

This section defines the aforementioned terms of *resource-constrained* and *embedded system* more precisely. After introducing basic terminology and giving necessary background, we proceed to elaborate data collection methods and introduce the employed evaluation platforms.

2.3.1 Embedded Systems

An embedded system is a combination of computer hardware and software, often with additional mechanical or electrical components, designed to perform a dedicated function. It is different to general-purpose computers (such as personal computers) as its hardware and software are designed specifically for this dedicated function. Your computer mouse and keyboard are as much of embedded devices as your alarm clock, microwave oven or the urine tank on the international space station. Embedded systems are typically characterized by constraints of the technical context they are embedded into. Common design requirements include processing power, memory, power consumption, reaction time, reliability, lifetime guarantees, production cost and development costs. This affects the *embedded developer* in that he has to develop the system to use as few resources as possible to design a competitive product. Which is why embedded software is usually so specialized, that it cannot be transferred to other systems. Collecting relevant data points during research therefore has to be done with caution. As embedded systems tend to be so different from each other, only certain observations within experimental setups provide insights that can be generalized.

In this thesis we focus on the engineering requirements of processing power, memory consumption and reaction time. Processing power and reaction time are closely related, but do not always strongly correlate, as in some embedded scenarios the reaction time is mainly determined by the size of transmitted data or the employed transmission medium. Similarly, power consumption is related to processing power as well. However, often times the power consumption of the embedded system's processor is negligible compared to that of its communication peripherals or actuators. Revealing specific insights about the power consumption of an experimental setup is therefore not generalizable and hence omitted.

Components of an Embedded System

At the heart of an embedded system is a microcontroller. A microcontroller unit (MCU) is an integrated system that includes a microprocessor, both volatile and non-volatile memory and necessary peripherals. In contrast to most general-purpose computers, MCU microprocessors typically follow the Harvard architecture, meaning they fetch instructions and data from separate memory. Executable instructions are commonly stored in some form of Read-Only Memory (ROM), whereas dynamic data is stored in Random-Access Memory (RAM). Both are described further in Section 2.3.2.

From a software perspective, embedded systems can vaguely be distinguished into two families: embedded and deeply embedded. Embedded software that does not require an embedded operating system (eOS) component is sometimes referred to as "deeply embedded" or "bare metal". Although the line between those two families is blurry, the distinction can be useful for interpreting results later in this thesis. Part II of the thesis deals with experiments conducted in embedded systems, whereas experiments in Part I are partially conducted "bare metal" / deeply embedded.

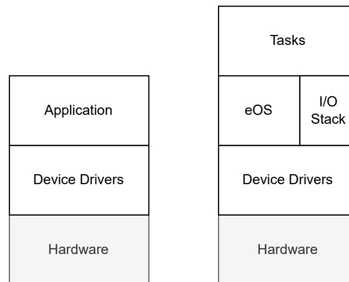


Figure 2.2: Diagram of typical deeply embedded (left) and embedded (right) software stacks.

Figure 2.2 shows the two typical software stacks for embedded systems. The main difference regarding experiments is that the operating system introduces noise into measurements, as it handles interrupts and conducts certain housekeeping tasks. Further noise stems from the operating system's I/O operation, e.g. communication with a network peripheral. How this is handled in our experiments is described in Section 2.3.5.

2.3.2 Key Metrics in Embedded Systems

As previously mentioned, this thesis is mainly concerned with how post-quantum cryptography impacts requirements regarding processing power, memory consumption and reaction time. In this section we define these terms more precisely and give necessary background.

Processing Power

Processing power is the capacity of a system to process data and execute operations. For a given single-core embedded system, it can be altered in two ways. First, additional hardware could be included, such as a floating point unit into the microprocessor or an external cryptography accelerator into the MCU. This would raise costs, but might accelerate certain computations. Secondly, the microprocessor's base clock frequency can usually be altered to some degree. Higher clock frequencies lead to higher processing power, but also to more power consumption. To gather meaningful results, the processing power employed during an experiment has to be detailed.

Memory

Memory as a resource has two meanings. Commonly, embedded systems have volatile and non-volatile memory. The non-volatile memory retains its contents even when power is removed. Traditionally, these memory types are referred to as ROM (non-volatile) and RAM (volatile). However, with modern memory technology the line between the two has blurred. Non-volatile memory is mostly used to store the application code, that is the instructions to be executed, and additional resources necessary to execute the code. Pre-installed certificates that form a root of trust could, e.g., be such a resource. ROM refers to a whole family of memory with *masked* ROM that has its content "burned in", *programmable* ROM (PROM) that can be written once, *erasable-and-programmable* ROM (EPROM) that can be re-written to using specialized hardware and *electrically-erasable-and-programmable* ROM (EEPROM) that can be re-written electrically. Commonly, masked ROM is used for inexpensive embedded devices that are produced in large quantities (e.g. an alarm clock). Other types of embedded devices increasingly make use of flash memory. This type of memory is also non-volatile but can be re-written many times and offers faster reading times at lower costs than EEPROMs. Often times, flash memory is *locked* for writing during the operation of the embedded device and only made writeable when software needs to be updated. Confusingly, this is why flash memory is also sometimes referred to as ROM. In this thesis we refer to non-volatile memory as ROM, independent of the employed memory technology.

Volatile RAM memory on the other hand, stores dynamic data the system generates or operates on during runtime. Static RAM (SRAM) preserves its data as it is provided with power. Dynamic RAM (DRAM) on the other hand holds data for only a very brief period, typically far under a second, when powered. DRAM therefore has to be combined with a DRAM controller that refreshes its content frequently. SRAM is typically much faster than DRAM, but also more expensive. Within our experiments, only MCUs with SRAM are employed. Specifics regarding ROM and RAM technologies and usage in our experiments are further detailed in Section 2.3.5.

Reaction Time

Reaction time refers to the time between an event (e.g. sensor signal or incoming network data) and the embedded system's response. This reaction time is influenced by the employed algorithms, processing power, memory access speed and in networked embedded systems it is critically affected by transmission size, bandwidth and latency. Bandwidth refers to the maximum data transfer rate of a communication channel, while latency is the time it takes for a data packet to travel from source to destination.

2.3.3 Cortex-M Processor Family

All MCUs employed throughout this thesis use Arm Cortex-M3 or Arm Cortex-M4 processors. Arm (previously Advanced RISC Machines Ltd) is a company that designs processors and licenses those designs to processor manufacturers. The Arm Cortex-M family is a range of 32-bit processor core designs optimized for low-power, cost-sensitive embedded applications. Cortex-M3 and Cortex-M4 processors implement the very similar `Armv7-M` (M3) and `Armv7E-M` (M4) instruction set architectures (ISAs). We choose the lower-cost M3 when feasible for an embedded use case. The Cortex-M4 is the NIST-recommended architecture for evaluating post-quantum cryptography in embedded settings. For this reason, many Cortex-M4-optimized implementations of post-quantum algorithms exists. The PQM4 repository [KRSS], for example, curates a large collection of such implementations. In addition to the Cortex-M3 features, the Cortex-M4 platform offers single-cycle multiplication, basic single-instruction multiple-data (SIMD) instructions and an optional floating point unit (FPU). All of this can be used to improve certain types of post-quantum algorithms. An M4 processor including the optional FPU is commonly referred to as Cortex-M4F. Both the Cortex-M3 and M4 are somewhat mid-range of the Cortex-M processor family. They are more capable than the ultra-low-power M0, M0+ and M23 cores but offer less computational power than the M7, M33, M55 or M85 cores and therefore offer a good trade-off for evaluation of embedded use cases.

2.3.4 Embedded Software

Developers of embedded software need to "get their hands dirty", as they have to interact with hardware and work within the boundaries of their limited platform. To get this job done, the vast majority of embedded devices is programmed in the C or C++ programming languages, with occasional assembly-routines for example used as interrupt service routines (ISRs). Using a high-level language like C/C++ instead of assembly has the advantage that development is faster and hardware-independent routines can be used across platforms. This portability of code can be very advantageous over time as hardware requirements might change. An advantage of C and C++ as high-level languages in particular, is that they are comparatively "low level". They allow developers

to read and write from arbitrary addresses and use pointer arithmetic to directly access computer objects. With memory-mapped peripherals, this leads to high-level code interacting with the hardware. Additionally, they are fairly simple to learn and compilers are available for pretty much every processor architecture. On the downside, also memory management is left up to the developer, which is a common cause for memory corruption vulnerabilities.

The Rust programming language is a fairly new development. It is memory safe and is, in principle, not affected by this vulnerability class. Just like C it is compiled to *native* code and does not require garbage collection. For its memory safety, ease of use and modern tooling it has seen steady adoption in the embedded world. Experiments detailed within this thesis use software written in either C or Rust, with the exception of some cryptographic primitives written in assembly.

Within the thesis, certain compiler flags are displayed to show in which dimension the code was optimized by the compiler. The flags used are identical for both C and Rust compilers and displayed in Table 2.3.

Table 2.3: Common compiler flags used throughout this work.

Flag	Description
-*Oz*	Optimize for smallest possible binary size – saves ROM.
-*O0*	Do not optimize – is useful for debugging.
-*O3*	Optimize for speed – improves reaction times.
-*flto*	Use link-time optimization – improves size and speed.

While flags like -*O3* usually lead to faster code, they also lead to larger code requiring more ROM space, e.g., due to code unrolling. Link-time optimization performs cross-file optimizations at link time and generally comes with no downsides except for longer compilation times.

2.3.5 Evaluation Platforms and Data Collection

All experiments documented in this thesis were conducted using evaluation boards of large MCU manufacturers. These evaluation boards have multiple advantages over maker-style systems such as Arduino or Raspberry Pi. First and foremost, these boards are designed for original equipment manufacturers (OEMs) to test the MCUs under realistic conditions for specific embedded use cases. Consequently, these evaluation boards come with extensive documentation about their exact inner workings and characteristics, down to precise timings of internal components and buses, whereas maker-style board documentations focus on how to use the board, as-is, in a project. Because the MCU manufacturers want evaluation boards to be appealing to OEMs, they are also less costly and readily available, enabling other researchers to reproduce results or build on them. Table 2.4 shows key characteristics of the evaluation boards employed in this thesis.

Table 2.4: Overview of utilized evaluation boards with their prime characteristics: CPU architecture & clock frequency, SRAM and flash capacity. Communication peripheral employed during experiments are specified under i/o port.

Board		**CPU**		**Memory**		**Periphery**
Producer	**Model**	**Arch.**	**Freq.** (MHz)	**RAM** (kB)	**ROM** (kB)	**i/o port**
Silicon Labs	STK3701A	M4F	72	512	2 048	RJ-45
Nordic	NRF52840	M4F	64	256	1 024	NFC,USB
STM	F207ZG	M3	120	128	1 024	UART

Experimental data was gathered for ROM and RAM usage, run and reaction times, transmission sizes and transmission times. These characteristics were collected in various ways, detailed in the following subsections.

Memory

Memory consumption has two dimensions that aren't necessarily independent of each other. Fewer ROM usage sometimes comes at the expense of more RAM usage and vice versa. These two characteristics are therefore measured and reported individually. Measuring ROM usage is straight forward. We measure ROM consumption as the number of bytes that will be written to flash memory before the evaluation board is started. Depending on the experiment, only a certain part of the actual code base will be measured (e.g., the TLS stack) to make comparisons easier. Measuring dynamic RAM usage is more difficult. First, it is dynamic, so it has to be measured on the board, and second, once there is an eOS or standard library involved, RAM is typically consumed in two ways: stack and heap. The heap is used for dynamic allocations of size unknown at compile time, for example in a TLS stack once it received headers containing object sizes. By convention, cryptographic primitives themselves should only consume stack space to avoid dependence on standard libraries and non-deterministic runtimes. In this work, stack usage is measured with a technique called *stack coloring*. It takes advantage of the fact that the stack grows in one direction and is reset to its original position after a routine ends. For stack coloring, a large enough portion of the stack is filled with a cookie value prior to executing the to-be-measured code branch. Once the measured code sequence is completed, a routine checks how many consecutive bytes on the stack do not contain the cookie value. This number represents the amount of stack space consumed during execution of the evaluated routine. Heap space on the other hand is allocated by the heap implementation, typically contained in the standard library. For our experiments, employed heap space is measured by including counters into the heap implementation or by using functionality included into the standard library/eOS for this purpose. In our results we add

up stack and heap usage into RAM usage results, unless stated otherwise.

Runtime and Reaction Time

Runtime and reaction time are presented either in CPU cycles or milliseconds. For measurements conducted on the board itself, the SysTick timer is employed. In Cortex-M processors, the SysTick timer is a 24 bit downward-counter register that can be latched to the system clock. Upon reaching the zero value, an interrupt is triggered. The ISR handling that interrupt then updates a global uptime counter. Using this timer therefore enables very precises measurements of runtime based on cycle counts. Converting cycle counts into time can be done trivially via multiplying the cycle count with the board's clock frequency. This works as the experiments only use MCUs at a fixed clock frequency. A problem when measuring runtime down to the cycle is the flash-based ROM. Flash is good for evaluation boards as it allows us to write different software onto the MCU. However, flash is much slower than masked ROM, which would likely be used in an actual low-cost embedded device after the evaluation phase is concluded. In case we want to measure algorithm runtime in a masked ROM setting, we therefore reduce the clock speed of our evaluation board to match that of the included flash, eliminating wait states. Runtimes are measured over many iterations and then averaged, sometimes with standard deviation specified, to minimize errors induced by outliers or eOS noise.

Transmission Size and Time

Transmission size of cryptographic objects is measured either in the I/O stack of the underlying eOS or, when communication is done with a computationally more powerful computer, within this computer. Transmission time on the board is measured equivalently to runtime, as described in the previous paragraph. Depending on the context, transmission time can either refer to runtime needed to write the data to the communication channel (e.g. a wireless channel before the device can power-down or idle again), or to establishing a connection that can then be used to transmit data (e.g., completing a TLS handshake). Specifics are detailed in the respective experiments.

Part I

Implementing Primitives

Chapter 3

Verifying Post-Quantum Signatures in 8 kB of RAM

The first chapter of this part is based on the publication:

> Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. Verifying Post-Quantum signatures in 8 kB of RAM. In *Post-Quantum Cryptography – PQCrypto 2021*, LNCS, pages 215–233. Springer, 2021. `https://eprint.iacr.org/2021/662`

In this chapter, we study implementations of post-quantum signature schemes on resource-constrained devices. We focus on verification of signatures and cover NIST PQC round-3 candidates Dilithium, Falcon, Rainbow, GeMSS, and SPHINCS$^+$. For this, we assume an ARM Cortex-M3 with 8 kB of memory and 8 kB of flash for code; a practical and widely deployed setup in, for example, the automotive sector. This amount of memory is insufficient for most schemes. Rainbow and GeMSS public keys are too big; SPHINCS$^+$ signatures do not fit in this memory. To make signature verification work for these schemes, we stream in public keys and signatures. When streaming the public key, the device needs to securely store a hash value of the public key to verify the authenticity of the streamed public key. During signature verification, the public key is incrementally hashed, matching the data flow of the streamed public key. Due to the memory requirements for efficient Dilithium implementations, we stream in the public key to cache more intermediate results. We discuss the suitability of the signature schemes for streaming, adapt existing implementations, and compare performance.

Some signature schemes presented in this work have been eliminated from the standardization process since the paper was published. As their performance results still carry value for future research, they remain in this chapter. However, eliminated schemes are clearly marked.

Contribution. We address the challenge of performing signature verification of post-quantum signature schemes with a large public key or signature in a highly memory-constrained environment. Our approach is to stream the public key or the signature. However, an alternative quantum-secure approach is presented as well. We show that this way signature verification can be done keeping only small data packets in constrained memory. We implemented and benchmarked the proposed public key and signature streaming approach for four different signature schemes (Dilithium, SPHINCS+, Rainbow, and GeMSS). Although for Dilithium streaming the public key is not strictly necessary, the saved bytes allow us to keep more intermediate results in memory. This results in a speed-up.

For comparison, we also implemented the lattice-based scheme Falcon for which streaming small data packets is not necessary in our scenario as the entire public key and signature fit into RAM. We demonstrate that the proposed streaming approach is very well suited for constrained devices with a maximum utilization of 8 kB RAM and 8 kB Flash.

Associated Software. The source code, featuring a benchmarking setup, communication library and steaming-aligned post-quantum implementations for reproducible results, is published under a permissive license and available at `http://doi.org/10.5281/zenodo.17381244`.

Organization of this chapter. After giving background on the research question of streaming PQC signatures in Section 3.1, we formally describe the investigated use case of feature activation and introduce an alternative solution employing only symmetric primitives in Section 3.2. In Section 3.3 we elaborate on how well-suited different families of post-quantum algorithms are for our use case and describe the specific approach to implementation. Implementation and benchmarking results are presented and discussed in Section 3.4.

3.1 Introduction

The generally larger keys and signatures of post-quantum signature schemes have enormous impact on cryptography on constrained devices. This is especially important when the payload of the signed message is much smaller than the signature, due to additional transmission overhead required for the signature. Such short messages are for example used in the real-world use case of feature activation in the automotive domain. Feature activation is the remote activation of features that are already implemented in the soft- and hardware of the car. For example, an additional infotainment package. Usually, a short activation code is protected with a signature to prevent unauthorized activation of the feature.

In the automotive sector, it is very common to perform all cryptographic operations on a dedicated hardware security module (HSM) that resembles a Cortex-M3 processor with a clock frequency of 100 MHz and limited memory resources, e.g., [HRS+09]. Typically, the HSM is in the same package as the

main processor with its own memory and is connected via an internal bus with a bus speed of about 20 Mbit/s. A fair estimate for available memory for signature verification on the HSM is under 18 kB of RAM and 10 kB of flash. However, we aim for a lower memory usage of 8 kB of RAM and flash to allow additional space for other applications and an operating system.

In this scenario signatures are verified in the very constrained environment of an HSM. It may not be able to store large public keys or keep large signatures in memory. Sometimes even the main processor does not have sufficient memory resources. Then the public key or signature must be provided to the HSM by another device in the vehicle network, like the head unit. In this case, the public key or signature must be streamed in portions over the in-vehicle network to the destination processor. A typical streaming rate over the CAN bus of an in-vehicle network is about 500 kbit/s, considering a low error transmission rate. Section 3.2.1 provides more details on the use case.

Related Work

To the best of our knowledge, this is the first work that addresses signature verification by streaming in the public key or signature. For signature schemes, streaming approaches have been investigated in [HRS16] but the focus of that work was on signature generation (for stateless hash-based signatures). The encryption scheme Classic McEliece was studied for constrained devices, solving the issue of public keys being larger than the available RAM by either streaming [RKK20, Str10] or placing them in additional Flash [CC21, EGHP09].

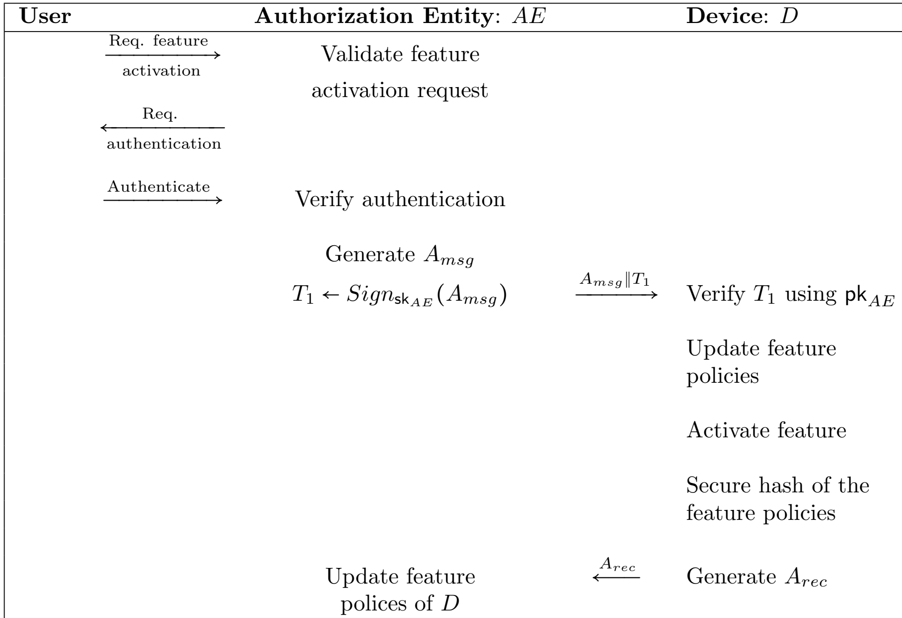
3.2 Background

This section gives background on the use-case of feature activation. Additionally, a proposed alternative protocol, exclusively based on symmetric primitives is presented.

3.2.1 Feature Activation

Feature Activation is intended to activate additional functionality on an embedded device that is already deployed and active in the running environment. It differs from a software update because all software and required hardware for the feature's functionality is already included in the device, but not activated.

The feature is activated by an authentic message from an authorized instance. The activation of the feature is device specific, therefore the activation messages must not be portable to other devices. Protocol 1 describes the actual feature activation process between an embedded device on which the feature is to be activated and an authorized instance, e.g., a back-end system. To authenticate the feature activation request, a signature is part of the message sent from the authorization instance to the embedded device. Nowadays, this signature is implemented, for example, by an ECC signature, which is not a



Protocol 1: Protocol for feature activation

post-quantum algorithm. In the scenario shown, the overall protocol does not take into account any resource constraints on the device, so that, for example, the ECC signature and the public key are stored entirely on the device.

Protocol 1 can be roughly paraphrased as follows: The user, e.g. the car owner, creates a request to activate a desired feature for a specific vehicle (identified by a vehicle identification number — VIN). This can be done through an online platform. The authorization instance, which can be represented by a back-end, validates the feature request for the feature policies it stores for the requested vehicle, and requests and verifies the user's authentication. Upon successful authorization, the authorization instance generates a device-specific feature activation request A_{msg} for the device that is part of the vehicle and implements the requested feature. Furthermore, the authorization instance generates a signature T_1 for the message A_{msg} using its private key sk_{AE}. When the device successfully verifies the signature T_1, it updates its feature policies, activates the requested feature, and stores the feature policy hash. Finally, the embedded device confirms the feature activation status in a message A_{rec} to the authorization instance. The authorization instance itself also updates and stores the feature policy for the specific device.

3.2.2 Alternative Implementation

For embedded applications it is sometimes attractive to use symmetric cryptography in place of public-key cryptography. Not only is symmetric cryptography a lot faster, it also benefits from already-present hardware acceleration and key sizes are significantly smaller. Of course, the secret keys in symmetric devices are extremely sensitive. We need that a secret key extracted from a particular deployed device does not compromise the entire scheme. This implies the need to provision each device with its own individualised key. However, when deploying hundreds or thousands of devices this means we have a potentially significant key management problem. Fortunately, the many-to-one architecture in this automotive application implies we only need a single key between each device and the back-end. Furthermore, each deployed device has public identifiers, like the vehicle VIN or a serial number. This allows us to only let the manufacturer store a single key for all deployed devices.

We use these properties to construct an efficient key distribution scheme. Let each device have a unique identifing number n. This could for example be the concatenation of the vehicle VIN and the device's serial number. We let the manufacturer generate a main secret key K_m. Then, we provision at time of manufacturing each device with the following key K_d, such that

$$K_d = \mathrm{KDF}(K_m, n).$$

Here, KDF is an appropriate key-derivation function.

Then, whenever the device needs to use their key K_d in communication with the manufacturer, they send over their identifier n along with the message. For example, if they need to send an authenticated message m, they might send $\{n, m, \mathrm{MAC}_{K_d}(n, m)\}$. The manufacturer can then easily compute K_d based on n and the main secret K_m, and verify the message. As the device does not have access to K_m, they can only have produced this MAC if they were provisioned with K_d at time of manufacture.

Of course, this entire scheme falls down when K_m is compromised. As such, special care needs to be taken to protect it. Although the private keys used in public-key cryptography also need to be protected, we can use revocation mechanisms to recover from a compromise. This is not possible with symmetric cryptography.

3.3 Analyzed Post-Quantum Signature Schemes

We now briefly discuss the different signature schemes considered and their adaptability to streaming. Our exposition is focused on signature verification due to limited space. For all schemes we selected parameters that meet at least NIST security level 1. Where possible we prioritized verification speed over signature speed as we assume that signatures are created on devices that are significantly more powerful than the ones we consider for signature verification.

3.3.1 Hash-Based Schemes

Hash-based signature schemes are signature schemes for which security solely relies on the security properties of the cryptographic hash function(s) used. In contrast to other proposals for digital signature schemes it does not require an additional complexity theoretic hardness assumption. Given that the security of cryptographic hash functions is well understood and even more, we know that generic attacks using quantum computers cannot significantly harm the security of cryptographic hash functions, hash-based signatures present a conservative choice for post-quantum secure digital signatures. The description in this section is simplified, and we refer to the official specification [ABB+20b] for a detailed exposition.

One-Time Signature Schemes (OTS)

Hash-based signatures build on the concept of a one-time signature scheme (OTS). This is a signature scheme where a key pair may only be used to sign one message. If two messages are signed with the same secret key, the scheme becomes insecure. Such OTS can be constructed from cryptographic hash functions. The very generic concept is that the secret key consists of random values while the public key contains their hash values. A signature consists of a subset of the values in the secret key, selected by the message. A signature is verified by hashing the values in the signature and comparing the resulting hash values to the respective values in the public key. The OTS commonly used today is the Winternitz OTS (WOTS) or variations thereof which generalize the above concept to hash chains. WOTS has the important property that a signature is verified by computing a candidate public key by hashing the values in the signature several times (depending on the message) and comparing the result to the public key.

Merkle Signature Schemes (MSS)

Given a OTS, a many-time signature scheme can be constructed following the concept of Merkle Signature Schemes (MSS) [Mer90]. For these, a number (a power of 2) of OTS key pairs is generated and their public keys are authenticated using a binary hash tree, called a Merkle tree. The hashes of the public keys form the leaves of the tree. Inner nodes are the hash of the concatenation of their two child nodes. The root node becomes the MSS public key. Assuming WOTS is used as OTS, a signature consists of the leaf index, a WOTS signature and the so-called authentication path (cf. Figure 3.1). The authentication path contains the sibling nodes on the path from the used leaf to the root. Verification uses the WOTS signature (and the message) to compute a candidate public key and from that the corresponding leaf. This leaf is then used together with the authentication path to compute a root node: Starting with the leaf, the current buffer is concatenated with the next authentication path node and hashed to obtain the next buffer value. The order of concatenation

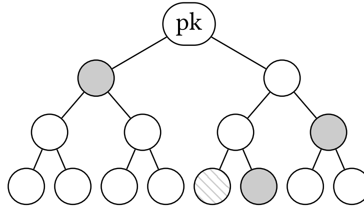


Figure 3.1: The authentication path of the fifth leaf (Source [BHK+19])

is determined by the leaf index in the MSS signature. The final buffer value is then compared to the root node in the public key.

In general, this leads a so-called stateful scheme as a signer has to remember which OTS key pairs she already used. This concept is the general idea underlying the schemes described in recent RFCs [MCF19, HBG+18] LMS and XMSS.

SPHINCS+

The limitation of having to keep a state as signer can be overcome in practice using the SPHINCS construction [BHH+15] (previous theoretically efficient proposals by Goldreich go back to the last century but were only of theoretical interest). SPHINCS+ [BHK+19] essentially instantiates the SPHINCS construction. The first idea in SPHINCS uses a few-time signature scheme (FTS) - a signature scheme where a key pair can be used to sign a small number of messages without keeping a state before the scheme gets insecure. SPHINCS+ uses a huge number of FTS key-pairs (in the order of 2^{64} depending on the parameters). For every new message, a random FTS key is picked to sign. By making the number of FTS keys large enough, the probability that one key gets used to sign more than a few messages can be made vanishingly small. The public keys of all these FTS key pairs are authenticated using a certification tree of MSS key pairs called the hypertree. The hyper tree is essentially a PKI. To the top MSS key works as a root CA and the bottom layer MSS keys certify FTS public keys. The whole structure is deterministically generated using pseudorandom generators. That way, it is not necessary to store which OTS keys where used for an MSS key because the message that a specific OTS key will be used to sign is predetermined.

The FTS in SPHINCS+ is FORS. A FORS secret key consists of several sets of random values the values in each set are authenticated via a Merkle tree. These trees have the hashes of the secret values as leaves. The public key is the hash of the concatenation of all root nodes of these Merkle trees. A signature consists of one secret key value from each set (determined by the message) and the respective authentication path. Verification works by computing the leaves from the signature values and afterwards computing candidate root nodes as for MSS. This can be done per tree. Afterwards, the roots are used to compute

a candidate public key.

A SPHINCS+ signature consists of a randomizer R that is hashed with the message to obtain the message digest, a FORS signature, and the MSS signatures on the path from the FORS keypair to the top tree. Verification computes a message digest using the message and R. The message digest is split into the index of the FORS signature and the indices of the Secret key values in the FORS signature. With this, the FORS signature is used to compute a candidate public key. This candidate FORS public key is used as message to compute a candidate MSS root node with the first MSS signature which is used as message for the next signature, and so on. The final MSS root node is compared to the root node in the public key.

3.3.2 Multivariate-Based Schemes

Multivariate signature schemes are based on the hardness of finding solutions to systems of equations in many variables over finite fields, where the degree of the equations is at least two. The first multivariate signature scheme was designed by Matsumoto and Imai [MI88] and broken by Patarin [Pat95]. Patarin with several coauthors went on to design modified schemes [Pat96, PGC98, KPG99] which form the basis of modern multivariate signature schemes.

To fix notation, let the system of equations be given by m equations in n variables over a finite field $\mathbb{F}_q$. Most systems use multivariate quadratic (MQ) equations, i.e. equations of total degree two. Then the m polynomials have the form

$$f_k(x_1, x_2, \dots, x_n) = \sum_{1 \le i \le j \le n} a_{i,j}^{(k)} x_i x_j + \sum_{1 \le i \le n} b_i^{(k)} x_i + c^{(k)} \tag{3.1}$$

with coefficients $a_{i,j}^{(k)}, b_i^{(k)}, c^{(k)} \in \mathbb{F}_q$.

Let M be a message and let $H : \{0,1\}^* \times \{0,1\}^r \to \mathbb{F}_q^m$ be a hash function. A signature on M is a vector $(X_1, X_2, \dots, X_n) \in \mathbb{F}_q^n$ and a string $R \in \{0,1\}^r$ satisfying for all $1 \le k \le m$ that $f_k(X_1, X_2, \dots, X_n) = h_k$ for $H(M, R) = (h_1, h_2, \dots, h_m)$. The inclusion of R is necessary because not every system has a solution.

Verification is conceptually easy – simply test that all signature equations hold. Signing depends on the type of construction and what information the signer has to permit finding a solution to the system.

General considerations for streaming

MQ systems lead to short signatures but the public keys need to contain the coefficients of (3.1) and are thus very large, in the range of a few hundred kB. The public keys can be streamed in in blocks of rows or columns, depending on how the public key is represented. At most m elements of $\mathbb{F}_q$ are needed to hold the partial results of evaluating $f_k(X_1, X_2, \dots, X_n), 1 \le k \le m$ in addition to the n elements for the signature and the m elements for the hash.

Rainbow

Rainbow [DCK+20] was a finalist in round 3 of the NIST competition. Rainbow uses two layers of the Oil and Vinegar (OV) scheme [Pat97]. For Rainbow the finite field is $\mathbb{F}_{2^4}$, so signatures require $\lceil m/2 \rceil$ bytes, leading to 66 bytes in NIST security level 1. We implement `rainbowI-classic` rather than one of the "circumzenithal" or "compressed" variants. Public keys are 158 kB for `rainbowI-classic`.

In Rainbow, the coefficients b and c are all zero. During verification, we load in columns $a_{i,j}^{(*)}$ corresponding to coefficients of each monomial $x_i x_j$, $i \leq j$. If $0 \neq x_i x_j = k \in \mathbb{F}_{16}$, we accumulate $a_{i,j}$ into a column $\mathbf{A}_k$, If $x_i = 0$, we skip all columns involving x_i. The final result is $\sum_{k \in \mathbb{F}_{16}^*} k\mathbf{A}_k$.

GeMSS

GeMSS [CFM+20] was an alternate in round 3 of the NIST competition. GeMSS is based on the HFEv− scheme [PCG01]. For GeMSS the finite field is $\mathbb{F}_2$, so signatures are very small, starting at 258 bits for category I, but to achieve security the public key needs to be very large, starting at 350 kB for category I.

It bears mentioning that GeMSS is special among multivariates in that it employs the Patarin-Feistel structure to achieve very short signatures, wherein a public key is used *four* times during the verification. With pubkey f being m equations in n variables, to verify the signature of the message $\mathbf{M}$, we do:

1. write the signature as $(\mathbf{S}_4, \mathbf{X}_4, \mathbf{X}_3, \mathbf{X}_2, \mathbf{X}_1)$ where $\mathbf{S}_i$ are size m and the $\mathbf{X}_i$ are size $n - m$ (so the actual length of the signature is $4n - 3m$).
2. At stage i, which goes from 4 to 1, we set $\mathbf{S}_{i-1} = f(\mathbf{S}_i \| \mathbf{X}_i) \oplus \mathbf{D}_i$, where $\mathbf{D}_i$ is the first m bits of $(\mathrm{SHA}-3)^i(\mathbf{M})$.
3. The signature is valid if $\mathbf{S}_0$ is the zero vector.

There are three types of GeMSS parameters. "RedGeMSS" uses very aggressive parameters; "BlueGeMSS" uses more conservative parameters. Just "GeMSS" falls in the middle, and this is what we choose to implement. The parameter set targeting NIST security level 1 is `gemss-128` and has 350 kB public keys.

3.3.3 Lattice-Based Schemes

Lattice-based cryptographic schemes are promising post-quantum replacements for currently used public-key cryptography since they are asymptotically efficient, have provable security guarantees, and are very versatile, i.e., they offer far more functionality than plain encryption or signature schemes.

Lattice-based signature schemes are constructed using one of two techniques, either the GPV framework that is based on the hash-and-sign paradigm [GPV08], or the Fiat-Shamir transformation [Lyu09]. The security

of lattice-based signature schemes can be proven based on hard lattice problems (usually the LWE problem, the SIS problem, and variants thereof) or the NTRU assumption.

Dilithium

Dilithium was a NIST round 3 finalist [BLD⁺20] and has since become standardized [Nat24b]. Signature verification for Dilithium works as follows: The public key $pk = (\rho, \mathbf{t_1})$ consists of a uniform random 256-bit seed ρ, which expands to the matrix of polynomials $\mathbf{A}$, and $\mathbf{t}_1$. For MLWE samples $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$, $\mathbf{t}_1$ is the first output of the Power2Round procedure [BLD⁺20, Figure 3], and $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}_q(\mathbf{t}, d)$ is the straightforward bit-wise way to break up an element $r = r_1 \cdot 2^d + r_0$, where $r_0 = r \mod 2^d$ and $r_1 = (r - r_0)/2^d$ with $-2^d/2 \leq r_0 < 2^d/2$. Hence, the coefficients of $\mathbf{t}_0$ are the d lower order bits and the coefficients of $\mathbf{t}_1$ — the second part of the public key — are the $\lceil \log q \rceil - d$ higher order bits of the coefficients of $\mathbf{t}$. To verify a signature $(\mathbf{z}, \mathbf{h}, c)$ for a message M, one computes $\mathbf{w}' = \mathbf{A}\mathbf{z} - c \cdot \mathbf{t_1} \cdot 2^d$, uses the hint vector $\mathbf{h}$ to recover $\mathbf{w}'_1 = \text{UseHint}(\mathbf{h}, \mathbf{w}')$, and finally verifies that $c = h(h(h(\rho \| t_1) \| M) \| \mathbf{w}'_1)$. For the details, we refer to [BLD⁺20].

All Dilithium parameter sets use $q = 2^{23} - 2^{13} + 1$ and $d = 13$. Hence, while the coefficients of $\mathbf{t}$ need 23 bits, the coefficients of the public key $\mathbf{t}_1$ need only 10 bits. We use the smallest instance of Dilithium, which is NIST level 2 parameter set `dilithium2`. The public key size of `dilithium2` in total is 1 312 bytes and a signature needs 2 420 bytes.

Falcon

Falcon, too, was a NIST round 3 finalist [PFH⁺20a] and is scheduled for standardization. Falcon's signature verification works as follows: A signature for message M, consisting of the tuple (r, s), can be verified given the public key $h = gf^{-1} \pmod q$, where $f, g \in \mathbb{Z}_q[x]/(\phi)$ for a modulus q and a cyclotomic polynomial $\phi \in \mathbb{Z}[x]$ of degree n. Firstly r and M are concatenated and hashed into a polynomial c and s is decompressed using a unary code into s_2. Then, $s_1 = c - s_2 h$ is computed and it is verified that (s_1, s_2) has a small enough norm $(\leq \lfloor \beta^2 \rfloor)$. Coefficients are compressed one-by-one and hence can be decompressed individually. The embedding norm that is computed in [PFH⁺20a, Algorithm 16, line 6] can be computed in linear time and only requires two coefficients at a time. However, the preceding polynomial multiplication requires all coefficients of one operand to be present, preventing coefficient-by-coefficient streaming for both the signature and the public key at the same time. If, however, only the signature or the public key is streamed, the polynomial multiplication could be performed. We use `falcon-512`, targeting NIST level I, which uses dimension $n = 512$ and $q = 12289 \approx 2^{14}$, hence each coefficient of the public key needs 14 bits.

3.4 Implementation

The following section describes the implementations of the signature schemes for the use case of feature activation described in Subsection 3.2.1. The signature verification is performed on a Cortex-M3. The consumption for program flash should be limited to 8 kB. The RAM usage should not exceed 8 kB. The bus speed for streaming is assumed to run at either 500 kbit/s or 20 Mbit/s.

3.4.1 Streaming Interface

Signed messages and public keys are streamed into the embedded Cortex-M3 device. To avoid performance overhead, our streaming implementation follows a very simple protocol. In a first step, the length of the signed message is transmitted to the embedded device. Then the embedded device initializes streaming by supplying a chunk size to the sender and additionally supplies if signed message or public key is to be streamed first. After every chunk, the embedded device can request a new chunk or return a verification result. The chunk size may be altered between chunks, but the public key and the signed message are always streamed in-order.The result is a one-bit message, signaling if the verification succeeded or failed, followed by the message in case the verification succeeded.

3.4.2 Public Key Verification

As the public key is being streamed in from an untrusted source, it is imperative to validate that the key is actually authentic. We assume that a hash of the public key is stored inside the HSM in some integrity-protected area.While the public key is being streamed in, we incrementally compute a hash of it that we eventually compare with the known hash. We use the same hash function as used by the studied scheme, i.e., SHA-256 for `sphincs-sha256` and `rainbowI-classic`, SHAKE-128 for `gemss-128` and SHAKE-256 for `dilithium2` and `falcon-512`. We keep the hash state in memory, occupying additional 200 bytes for SHAKE-128 and 32 bytes for SHA-256. We use the incremental SHA-256 and SHAKE implementations from pqm4 [KRSS19].

In the case of `gemss-128`, the public key is needed multiple times; once in every of the four evaluations of the public map. Note that the integrity needs to be verified each time.

3.4.3 Implementation Details

In the following, we describe the modifications to existing implementations of the five studied schemes needed to use them with the given platform constraints. Table 3.1 lists the public key, signature sizes, and the time needed for streaming them into the device at 500 kbit/s and 20 Mbit/s.

Table 3.1: Communication overhead in bytes and milliseconds at 500 kbit/s and 20 Mbit/s. GeMSS requires to stream in the public key *nb_ite* times (4 for `gemss-128`). All other schemes require streaming in the public key and signed message once.

	streaming data			streaming time	
	$\|pk\|$	$\|sig\|$	total	500 kb/s	20 Mb/s
`sphincs-s`[a]	32	7 856	7 888	126.2 ms	3.2 ms
`sphincs-f`[b]	32	17 088	17 120	273.9 ms	6.9 ms
`rainbowI`[c] †	161 600	66	161 666	2 586.7 ms	64.7 ms
`gemss-128` †	352 188	33	1 408 785[d]	22 540.6 ms	563.5 ms
`dilithium2`	1 312	2 420	3 732	59.7 ms	1.5 ms
`falcon-512`	897	690	1 587	25.4 ms	0.6 ms

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple` [c] `-classic` [d] $4 \cdot |pk| + |sig|$

†: Scheme was eliminated from the NIST standardization project.

SPHINCS⁺

Our SPHINCS⁺ implementation is based on the round-3 reference implementation [ABB⁺20b]. Preceding work [KRSS19] shows that computation time for SPHINCS⁺ verification on single-core embedded devices is almost exclusively spent in the underlying hash function. We did therefore not investigate further optimization possibilities. Aligning the implementation to a streaming API is fairly straightforward as SPHINCS⁺ signatures get processed in-order. For both `sphincs-sha256-128f-simple` and `sphincs-sha256-128s-simple` a public key is 32 bytes and hence does not require streaming.

For `sphincs-sha256-128f-simple`, a signature is 17 088 bytes. Contents of such a signature are displayed in Figure 3.2. The selected SPHINCS⁺ parameter sets use $n = 16$ byte and so a streaming chunk size of 16 bytes is possible. However, such a small chunk is undesirable due to overhead in terms of memory and computation. The leading 16 bytes of the signature make up the randomizer value, followed by the 3 696 byte FORS signature, consisting of 33 authentication paths, and the 13 376 bytes for 22 MSS signatures. Our implementation first processes a 3 712 byte chunk containing the randomizer value and FORS signature. This is used to compute the message digest and then the FORS root, evaluating the 33 authentication paths. Then, the computed FORS root is verified using the MSS signatures. The overall 22 MSS signatures, each consisting of a WOTS+ signature and an authentication path, are processed in three chunks. Given the memory constraints, the largest available chunk size is 4 864 bytes containing 8 MSS signatures. MSS signature streaming is therefore done in two 4 864 byte chunks and one final 3 648 byte chunk. Starting from the FORS root, this data is used to successively reconstruct all the MSS tree roots from the respectively previous root: first computing 67 hash chains using the WOTS+ signature, compressing their end nodes in a single hash, and then evaluating an authentication path. The last (or "highest") MSS tree root is

then compared to the root node in the public key. For this to work, the reserved chunk buffer needs to be 4 864 bytes large.

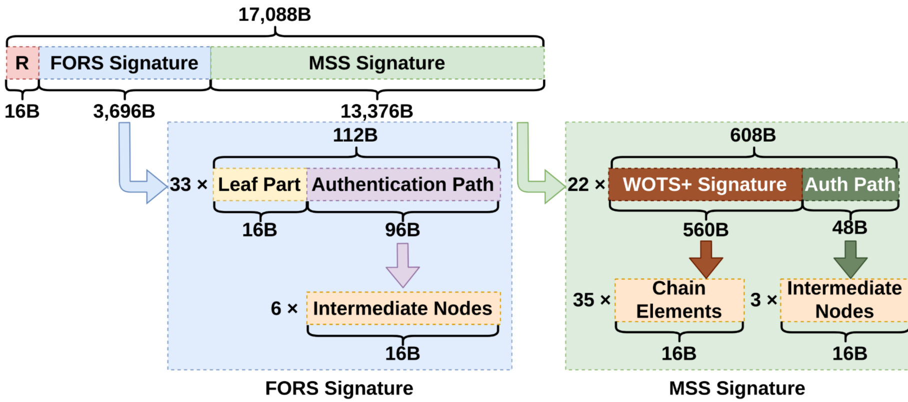


Figure 3.2: A `sphincs-sha256-128f-simple` signature unrolled.

For `sphincs-sha256-128s-simple`, streaming works analogously. The signature size is 7 856 bytes and only seven - slightly larger - MSS signatures are used within the scheme. This makes it possible to stream in all 7 MSS signatures in a single 4 928 bytes chunk. Streaming therefore consists of one FORS+randomizer-value (2 928 bytes) and one MSS (4 928 bytes) chunk.

Rainbow

The round-3 submission of Rainbow [DCK+20] contains an implementation targeting the Cortex-M4. As it relies only on instructions also available on the Cortex-M3, it is also functional on the Cortex-M3. However, due to the large public key (162 kB), we adapt the implementation for streaming. Rainbow signatures consist of an ℓ bit (128 for `rainbowI-classic`) salt and n (100) variables x_i in a small finite field ($\mathbb{F}_{16}$ for `rainbowI-classic`). Two $\mathbb{F}_{16}$ elements are packed into one byte in the signature and public key. We first unpack the elements of the signature and store one x_i in the lower four bits of a byte. This doubles memory usage from 50 to 100 bytes, but makes look-ups for individual elements easier. After the signature and corresponding x_i are stored in memory, the public key is streamed in. The public key consists of the Macaulay matrix $p_{i,j}^{(k)}$ representing the public map consisting of m (64) equations of the form

$$p^{(k)}(x_1, \dots, x_n) = \sum_{i=1}^{n} \sum_{j=i}^{n} p_{i,j}^{(k)} x_i x_j$$

with the x_i, x_j corresponding to the variables from the signature. For computational efficiency the public key is represented in the column-major form. The public key's first 32 byte chunk therefore has the form $[p_{1,1}^{(1)} | p_{1,1}^{(2)} | \dots | p_{1,1}^{(m)}]$

and the contained coefficients should be multiplied by x_1^2. Subsequent 32 byte chunks have the same form, i.e. $([p_{1,2}^{(1)}|\dots|p_{1,2}^{(m)}]$ should be multiplied by $x_1 \cdot x_2$ and so forth. Figure 3.3 visualizes such a public key and signature. To increase

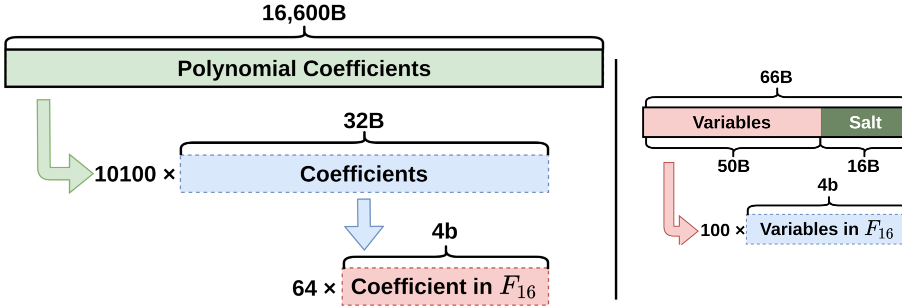


Figure 3.3: A `rainbowI-classic` public key (left) and signature (right) unrolled.

performance, Rainbow implementations delay multiplications. Before the actual multiplication step, coefficient sums are accumulated. Every incoming 32 byte chunk is added to one of 15 accumulators a_k based on the 15 possible values of $y_{i,j} = x_i \cdot x_j$ with $y_{i,j} > 0$. If $y_{i,j}$ is zero, the chunk is discarded. Once all chunks are consumed, every accumulator is multiplied by its corresponding factor $\tilde{a}_k = [k \cdot a_k^{(1)}|k \cdot a_k^{(2)}|\dots]$ and added summed up the final result.

One can exploit that if an element x_i is zero, all monomials $x_i x_j$ will be zero and the corresponding columns of the public key will not contribute to the result. As every 16th x_i is expected to be zero, this results in a significant speed-up. As Rainbow is using $\mathbb{F}_{16}$ arithmetic, additions are `XOR`. For multiplications, we use the bitsliced implementation from the Rainbow Cortex-M4 implementation [DCK+20].

The smallest reasonable chunk size for Rainbow is a single column of the Macaulay matrix, i.e., 32 bytes. However, as larger chunk sizes result in lower overhead, we use the largest chunk size which fits in our available memory. Due to the low memory footprint of the Rainbow implementation, we can afford to use chunks of 214 columns, i.e., 6 848 bytes. As there are 5 050 $(n \cdot (n+1)/2)$ columns, the last chunk is only 128 columns, i.e., 4 096 bytes.

GeMSS

To the best of our knowledge, there are no GeMSS implementations available targeting microcontrollers and we, hence, write our own. We base our GeMSS implementation on the reference implementation accompanying the specification [CFM+20]. The biggest challenge is that the entirety of the 352 kB public key is needed in each of the evaluations of the public map $\mathbf{p}$. Due to the iterative construction of the HFEv- scheme, there appears to be no better approach

than streaming in the public key in each iteration, i.e., *nb_ite* (4 for `gemss-128`) times.

Each application of the public map p requires the computation of

$$p_i = \sum_{i=0}^{n+v} \sum_{j=i}^{n+v} x_i x_j a_{i,j} + a_0.$$

Each column of the Macaulay matrix needs to be multiplied by a product of two variables and then added to the accumulator. The field used by GeMSS is $\mathbb{F}_2$ and, hence, field multiplication is logical `AND` and field addition is `XOR` which allows straightforward bitslicing of operations. Unfortunately, since the number of equations (m) is not a multiple of 8 ($m = 162$ for `gemss-128`), one cannot simply store the Macaulay matrix in column-major form since this would result in the columns not being aligned to byte boundaries. Therefore, GeMSS stores the first $8 \cdot \lfloor m/8 \rfloor$ (160) equations in a column-major form making up the first $\lfloor m/8 \rfloor \cdot (((n+v) \cdot (n+v+1))/2+1)$ (347 840) bytes of the public key with $n+v$ ($n = 174, v = 12$) being the number of variables. The last 2 equations are stored row-wise occupying the last $2 \cdot (((n+v) \cdot (n+v+1))/2+1)/8$ (4348) bytes.

We split the computation in two parts: The first $8 \cdot \lfloor m/8 \rfloor$ equations and the last ($m \mod 8$) equations. For the former, the most important optimization comes from the observation that if either of the two variables x_i or x_j is zero, the corresponding column does not impact the result. Similar to the Rainbow implementation, in the case x_i is zero, the entire inner loop and, hence, $n+v-i$ columns of the public key can be skipped. As half of the x_i are expected to be zero, this results in a vast performance gain. For the last ($m \mod 8$), we first compute the monomials $x_i x_j$ and store them in a vector, then we add this vector to each row of the public key. Lastly, we compute the parity of each row. The smallest reasonable chunk size for the first part of the computation is one column of the public key (20 bytes), while it is one row (2174 bytes) for the second part. However, we use 4 560 byte-chunks (285 columns) to achieve lowest overhead with 8 kB RAM.

Dilithium.

Our Dilithium implementation is based on previous work targeting the Cortex-M3 and Cortex-M4 [GKS20]. However, this work predates the round 3 Dilithium submission [BLD⁺20] which introduced some algorithm tweaks and parameter changes. Most notably for the performance of `dilithium2` verification, the matrix dimension of $\mathbf{A}$ changed from $(k, \ell) = (4,3)$ to $(4,4)$. Therefore, we adapt the existing Cortex-M3 implementation to the new parameters.

For `dilithium2`, the implementation of [GKS20] requires 9 kB of stack in addition to the 2.4 kB signature and 1.3 kB public key in memory. We apply a couple of tricks to fit it within 8 kB: We compute one polynomial of $\mathbf{w}'$ at a time, which allows us to stream in the public key $\mathbf{t_1}$.

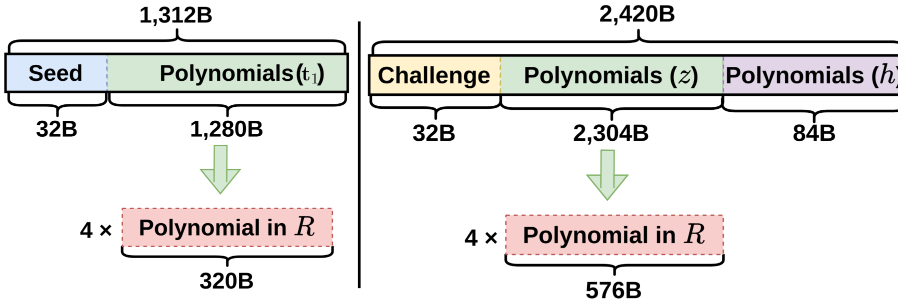


Figure 3.4: A `dilithium2` public key (left) and signature (right) unrolled.

Usually, one computes $\mathbf{w}' = \mathbf{A}\mathbf{z} - c \cdot \mathbf{t_1} \cdot 2^d$ as $\texttt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \texttt{NTT}(\mathbf{z}) - \texttt{NTT}(c) \cdot \texttt{NTT}(\mathbf{t_1} \cdot 2^d))$. Hence, it is desirable for performance to keep $\texttt{NTT}(\mathbf{z})$ and $\texttt{NTT}(c)$ in memory. However, that already occupies 5 kB. We instead keep the compressed forms of $\mathbf{z}$ and $\mathbf{c}$ in memory, occupying only $\ell \cdot 576 = 2304$ and 32 bytes, respectively, and recompute the NTT operations.

Previous implementations of Dilithium use 3 temporary polynomials to compute $\texttt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \texttt{NTT}(\mathbf{z}) - \texttt{NTT}(c) \cdot \texttt{NTT}(\mathbf{t_1} \cdot 2^d))$, one for the accumulator and two temporary ones for the inputs. We instead compute $\texttt{NTT}^{-1}(-\texttt{NTT}(c) \cdot \texttt{NTT}(\mathbf{t_1} \cdot 2^d) + \hat{\mathbf{A}} \cdot \texttt{NTT}(\mathbf{z}))$, which can be computed in 2 polynomials by sampling $\hat{\mathbf{A}}$ coefficient-wise, as was also proposed for Kyber [BKS19].

The total memory consumption comprises the 2 420-byte signature, 2 polynomials of 1 024 bytes each, 3 Keccak states of 200 bytes each, and about 600 bytes of other buffers, i.e., approximately 5 670 bytes in total. To improve speed, one can cache as much of $\texttt{NTT}(\mathbf{z})$ and $\texttt{NTT}(\mathbf{c})$ as possible. We cache $\texttt{NTT}(\mathbf{c})$ and 3 polynomials of $\texttt{NTT}(\mathbf{z})$ while still remaining below 8 kB of stack.

Falcon

We used a Cortex-M4 optimized implementation which is also part of Falcon's round-3 submission [Por19, PFH+20a]. It is compatible with Cortex-M3 processors, but relies on emulated floating point arithmetic. This leads to data-dependent runtimes, which is unproblematic for verification, but may be an issue when considering signing as well. On the Cortex-M3, the implementation submitted to NIST uses around 500 bytes of stack space, public keys of circa 900 bytes, signatures of around 800 bytes, and a 4 kB scratch buffer. The overall memory footprint is about 6.5 kB. Hence, streaming in the data in small packets is not necessary. Our implementation copies the whole public key and signature to RAM before running the unmodified Falcon verification algorithm.

Table 3.2: Cycle count for signature verification for a 33-byte message. Average over 1 000 signature verifications. Hashing cycles needed for verification of the streamed in public key (hashing and comparing to embedded hash) are reported separately. We also report the verification time on a practical HSM running at 100 MHz and also the total time including the streaming at 20 Mbit/s.

	w/o pk vrf.	w/ pk verification			w/ streaming
		pk vrf.	total	time[e]	20 Mbit/s
`sphincs-s`[a]	8 741k	0	8 741k	87.4 ms	90.6 ms
`sphincs-f`[b]	26 186k	0	26 186k	261.9 ms	268.7 ms
`rainbow`[f] †	333k	6 850k[d]	7 182k	71.8 ms	136.5 ms
`gemss-128` †	1 619k	109 938k[c]	111 557k	1 115.6 ms	1 679.1 ms
`dilithium2`	1 990k	133k[c]	2 123k	21.2 ms	21.8 ms
`falcon-512`	581k	91k[c]	672k	6.7 ms	8.2 ms

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple` [c] SHA-3/SHAKE [d] SHA-256
[e] At 100 MHz (no wait states) [f] `-classic`
†: Scheme was eliminated from the NIST standardization project.

3.5 Results

We chose an ARM Cortex-M3 board with 128 kB RAM and 1 MB Flash, an STM32 Nucleo-F207ZG, as the platform for the implementation of our case study. This board meets most of the specifications of an environment with limited resources of a typical automotive HSM embedded in MCUs. The only mismatch is the non-volatile memory (NVM). A typical limited HSM has much less NVM.

We clock the Cortex-M3 at 30 MHz (rather than the maximum frequency of 120 MHz) to have no Flash wait states. In a practical deployment in an HSM one would use fast ROM instead of Flash and, hence, our cycle counts are close to what we would expect in an automotive HSM.

We base our benchmarking setup on the pqm3[1] framework and adapt it to support our streaming API. For counting clock cycles, we use the SysTick counter. We stream in the signed message and public key using USART, but disregard the cycles spent waiting for serial communication. We stream in the signed message and public key using USART using a baud rate of 57 600 bps, which is much slower than what we would expect in a practical HSM. We use `arm-none-eabi-gcc` version 10.2.0 with `-O3`. We use a random 33-byte message which resembles the short messages needed for feature activation.

Table 3.2 presents the speed results for our implementations. The studied signature schemes rely on either SHA-256 (`rainbowI-classic`, `sphincs-sha256`) or SHA-3/SHAKE (`dilithium2`, `falcon-512`, and `gemss-128`). In a typical HSM-enabled device SHA-256 would be available in hardware and SHA-3/SHAKE will also be available in the future. However,

[1] `https://github.com/mupq/pqm3`

on the Nucleo-F207ZG no hardware accelerators are available. Hence, we resort to software implementations instead. For SHA-256 we use the optimized C implementation from SUPERCOP.[2] For SHA-3/SHAKE, we rely on the ARMv7-M implementation from the XKCP.[3]

While GeMSS and Rainbow only compute a (randomized) hash of the message, SPHINCS+, Dilithium, and Falcon use hashing as a core building block of the verification. Consequently, the amount of hashing in multivariate cryptography is minimal (2% for `rainbowI-classic`, 4% for `gemss-128`), while it makes up large parts for lattice-based (65% for `dilithium2`, 36% for `falcon-512`) and hash-based signatures (90% for `sphincs-sha256-128s-simple` and 88% for `sphincs-sha256-128f-simple`). Clearly lattice-based and hash-based schemes would benefit more from hardware accelerated hashing.

Additionally, we need to verify the authenticity of the streamed in public key. We report the time needed for public key verification separately. For hash-based signatures this operation comes virtually for free as the public key itself can be stored in the device, so that no hashing is required. For multivariate cryptography, the public key verification becomes the most dominant operation due to the large public keys and fast arithmetic. This is particularly pronounced for GeMSS as the public key is the largest and needs to be verified 4 times.

Table 3.3 presents the memory requirements of our implementations.

Table 3.3: Memory and code-size requirements in bytes for our implementations. Memory includes stack needed for computations, global variables stored in the .bss section and the buffer required for streaming. Code-size excludes platform and framework code as well as code for `SHA-256` and `SHA-3`.

	memory				code
	total	buffer	.bss	stack	.text
`sphincs-s`[a]	6 904	4 928	780	1 196	2 724
`sphincs-f`[b]	7 536	4 864	780	1 892	2 586
`rainbowI-classic` †	8 168	6 848	724	596	2 194
`gemss-128` †	8 176	4 560	496	3 120	4 740
`dilithium2`	8 048	40	6 352	1 656	7 940
`falcon-512`	6 552	897	5 255	400	5 784

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple`

†: Scheme was eliminated from the NIST standardization project.

[2] `https://bench.cr.yp.to/supercop.html`

[3] `https://github.com/XKCP/XKCP`

Chapter 4

Formally Verified Zeroization for Secret Values in RAM

This chapter is based on work published in:

> Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, and Peter Schwabe. High-assurance zeroization. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):375–397, 2024. `https://eprint.iacr.org/2023/1713`

This chapter introduces a solution that can benefit implementation of post-quantum, pre-quantum and symmetric primitives alike. In it, we revisit the problem of erasing sensitive data from memory and registers during return from a cryptographic routine. While the problem and related attacker model is fairly easy to phrase, it turns out to be surprisingly hard to guarantee security in this model when implementing cryptography in common languages such as C/C++ or Rust. We revisit the issues surrounding zeroization and then present a principled solution in the sense that it guarantees that sensitive data is erased and it clearly defines when this happens. We implement our solution as extension to the formally verified Jasmin compiler and extend the correctness proof of the compiler to cover zeroization. We show that the approach seamlessly integrates with state-of-the-art protections against microarchitectural attacks by integrating zeroization into Libjade, a cryptographic library written in Jasmin with systematic protections against timing and certain microarchitectural attacks. We present benchmarks showing that in many cases the overhead of zeroization is barely measurable.

Contribution. The main contribution of this work is presented in Section 4.4. We describe our principled approach to stack (and register) zeroization for the

Jasmin framework for high-assurance cryptography [ABB+17, ABB+20a]. The three main components of the Jasmin framework are its programming language, which can be used for writing high-speed implementations using the "assembly in the head" paradigm, its compiler, which comes with formal proofs of functional correctness and of preservation of side-channel protection (specifically preservation of constant-time), and its verification infrastructure, which can be used to prove functional correctness, constant-timeness, and reductionist security via an embedding to EasyCrypt. The Jasmin framework has been used for writing efficient and formally verified implementations of several key cryptographic primitives; in particular, it has been used to develop the Libjade library for post-quantum cryptography.

In this work, we use two main properties of Jasmin. First, Jasmin programs have a well-defined interface to outside callers (through `export` functions). Second, the Jasmin compiler can predict the stack usage of Jasmin programs at compile time. The latter is possible because, on the one hand, Jasmin programs are compiled as a whole, and on the other hand, for our applications, i.e. cryptographic primitives, programs do not use recursion. We use these two properties to leverage a compiler-based solution, which remains compatible with the global guarantees offered by Jasmin. In particular, we show that our modified compiler preserves correctness. In addition, we prove that zeroization integrates seamlessly with existing guarantees for constant-time and speculative constant-time [SBG+22].

Associated Software. A software artifact that enables readers to reproduce the benchmarks we present in Section 4.5 and the key-recovery from libsodium's ChaCha20 implementation we discuss in Section 4.2.3 is available at `http://doi.org/10.5281/zenodo.17381244`.

Organization of this chapter. In Section 4.2 we first motivate why zeroization is not as easy as one might think by identifying 3 different failure modes (plus the approach of explicitly excluding leakage of architectural state from the attacker model). We investigate how zeroization is handled in 11 popular open-source cryptographic libraries written in C or Rust. None of those libraries actually addresses the issue of memory zeroization with a principled and sound approach. We show that, for example, routines in libsodium and OpenSSL allow trivial key-recovery in our attacker model, despite attempts to perform some stack zeroization.

We then, in Section 4.3, review possible solutions both on the caller's side and on the callee side and discuss why those solutions are not widely implemented and used.

In Section 4.5 we show that the overhead of our protections is very small. Specifically, we extend Libjade [For23] from [SBG+22] with our protections against architectural-state leakage and present benchmarks showing that the cost is, for many routines, barely measurable, and remains solidly below 2% for all primitives except for highly optimized symmetric crypto routines on very short inputs. We conclude the chapter with a discussion of directions for future work in Section 4.6.

4.1 Introduction

Essentially all cryptographic software uses memory to temporarily store sensitive data during execution. Usually this memory is allocated on the stack, which means that when a cryptographic routine terminates and returns control to the caller, the memory becomes "invalid", but the sensitive contents remain. It is often mandated that such sensitive data in memory is erased or *zeroized* once it is no longer used. While overwriting data with zeroes seems like an easy task, it turns out that there are multiple failure modes and that many popular open-source crypto libraries actually do not have a sound approach to memory zeroization and in some cases can be shown to leave content in stack memory that allows trivial key recovery. This failure to perform zeroization has been noticed for specific libraries before; in this work, we systematize this observation by considering multiple libraries. The conclusion of our systematization is that unsound zeroization is a pervasive problem.

One might ask if this is an actual problem and what the reasons are to mandate zeroization – after all the memory is "invalid" after return. There are three reasons to consider an attacker, who obtains the stack contents after a cryptographic routine returns:

Certification. Certification of cryptographic software (or more generally "cryptographic modules") often requires zeroization of sensitive data. For example, the implementation guidance for FIPS 140-3 [Nat20] states in Section 9.7.A:

> *"A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys and CSPs within the module."*

Here, CSP stands for "critical security parameter". Similarly, the Common Criteria mandate in FCS_CKM_EXT.4(a) (*Cryptographic Key and Key Material Destruction*) that

> *"the TSF shall destroy cryptographic keys in accordance with a specified cryptographic key destruction method [selection: For volatile memory, the destruction shall be executed by a single direct overwrite [selection: consisting of a pseudo-random pattern using the TSF's RBG, consisting of a pseudo-random pattern using the host environment's RBG, consisting of zeroes] following by a read-verify[...]]",*

where TSF stands for the security functionality of the product under evaluation and RBG stands for "random bit generator".

Defense in depth. Cryptographic libraries are used by a plethora of applications and the application-level code is typically not under control of the library programmer. Bugs in the application that allow out-of-bound reads

of memory may be used to obtain memory content from the whole address space of the application, including, of course, all stack content. The most prominent example for such a bug is HeartBleed [Syn14, CVE14]. While this bug was part of the OpenSSL library (and not some application), it was not a bug in any of the cryptographic components. Zeroization helps to limit the damage that an attacker can do when exploiting such a vulnerability.

Forward secrecy. Finally, and most importantly, a principled approach to zeroization is required to guarantee forward secrecy. Many modern cryptographic protocols are built in such a way that compromise of long-term keys does not allow an attacker to recover plaintext of messages sent in the past; see, e.g., [Don17, Per18, Res18]. This is achieved by using only ephemeral keys to ensure confidentiality, which means that implementations need to ensure that those keys—and all information that allows recovery of these keys or plaintext encrypted under those keys—are erased from memory when they are no longer used. In other words, implementations need to ensure that keys are as ephemeral as specified in the protocol design.

For the remainder of this work we assume that there *are* good reasons to mandate memory zeroization. Specifically, we consider the following attacker model induced by mandating zeroization.

Attacker model. Throughout this chapter we consider an attacker who obtains leakage from a cryptographic computation. This leakage contains (possibly among other information) the content of all stack memory used by the cryptographic routine right after the routine returns control to the caller. We additionally include in the leakage the content of all registers and flags at the point of return from the cryptographic routine. We define a *cryptographic routine* to be an API function of a cryptographic library. This API function may call subroutines, but stack and register content is not leaked to the attacker every time such a subroutine returns. This reflects the idea that cryptographic computations need to "clean up" only when they return to a caller outside their control.

Note that this attacker model focuses on the core problem of zeroization and excludes several additional attack vectors that need to be addressed to prevent leakage of ephemeral sensitive data when a device is compromised. The most obvious attack vector is leakage through microarchitectural side channels. We will return to this later and show that zeroization composes seamlessly with state-of-the art (speculative) timing-leakage protection. In addition, arguments to cryptographic routines like encryption keys or plaintext are commonly owned by the caller and so it is the caller's responsibility to zeroize those arguments once they are no longer used. This is out of scope of our approach, which is focused on protecting cryptographic libraries rather than application code. Finally, while stack (and register) zeroization is essential to prevent data from

being written to disk as part of swapping or hibernation (aka "suspend to disk"), further measures are typically needed on system level to address this problem. For example, the `mlock` system call under Linux prevents memory regions from being swapped to disk and security-critical systems will probably want to disable hibernation altogether.

Related work. Multiple earlier works consider zeroization of sensitive information from different angles. Already in 2005, Chow, Pfaff, Garfinkel, and Rosenblum showed that it is generally not safe to rely on the fact that data will eventually be overwritten [CPGR05]. Their experiments show that *"data can remain in memory for days or weeks, even persisting across reboots"*. They introduce the "data lifetime" cycle:

> *"The span from first write to last read is the ideal lifetime. The data must exist in the system at least this long. The span from first write to deallocation is the secure deallocation lifetime. The span from first write to the first write of the next allocation is the natural lifetime. Because programs often rely on reallocation and overwrite to eliminate sensitive data, the natural lifetime is the expected data lifetime in systems without secure deallocation."*

Our attacker model roughly corresponds to the secure deallocation lifetime. Several similar discussions are found in the literature. In particular, Percival [Per14] reports on the challenges and shortcomings of tricking a compiler into zeroing buffers and stacks. Chapman [Cha17] provides another thorough discussion of why systematic zeroization is hard and proposes a path forward mostly in the context of the Ada and SPARK programming languages. Parts of the paper are akin to our discussion in Section 4.2 and also come to similar conclusions. Unlike our work, [Cha17] emphasizes the task of identifying what data in a program is sensitive; the approach we present here does not require this analysis and hence we only briefly discuss this in Section 4.6. None of these works presents an implementation of a solution to systematic zeroization.

Yang, Johannesmeyer, Olesen, Lerner, and Levchenko [YJO+17] provide a detailed analysis of the issue of dead-store elimination and of some popular scrubbing techniques; their analysis can be seen as much more in-depth review of what we briefly recall in Section 4.2.2. In addition, they provide a scrubbing function `secure_memzero` that combines different scrubbing techniques, and evaluated the use of the function with GCC, LLVM, and Visual C. Last, they implement a scrubbing safe dead-store elimination option in LLVM.

Our work is most closely related to [SCA18]. In that work, Simon, Chisnall, and Anderson first investigate how mainstream compilers make it actively hard to achieve certain security-related properties including zeroization. They then propose three approaches to perform zeroization in the LLVM compilers—all approaches are implemented and publicly available from `https://github.com/lmrs2/zerostack`. The first approach, coined function based, applies zeroization to all sensitive functions. The second approach, coined

stack-based, avoids zeroing the same stack area repeatedly. Their last approach, called call graph-based (**CGB**), is similar to what we have proposed for the Jasmin compiler. In particular, it does not support programs with unbounded recursion. As far as we know, their solution has unfortunately never been integrated into mainline LLVM. The main difference of our work is that we embed our solution into the Jasmin framework for high-assurance cryptography and are able to provide a formal proof of our approach. We briefly comment on the pros and cons of integrating zeroization in mainstream vs dedicated compilers in the conclusion.

Several works consider the challenges of memory scrubbing in the broader context of compiler-induced security bugs (CISB). These are discussed e.g. in [DPS15, XLD+23].

The problem of memory scrubbing has also been studied in the context of language-based security. Chong and Myers [CM05] propose a general language-based approach to let programmers express what data needs to be erased and a type system with multiple levels of confidentiality together with *declassification* and *erasure* operators. Hunt and Sands [HS08] study erasure as a noninterference property, and proposes a type system to check that erasure is performed, similar to [CM05]. Daniel, Bardin, and Rezk [DBR22] present a tool to analyze binaries, among other properties, for zeroization of sensitive data. These works do not consider the interactions between erasure and compilation.

Finally, there is a large body of work that formalizes the interactions between compilation and security. Many of these works focus on the theoretical underpinnings of these interactions, see e.g. [PAC19, ABC+21] for recent overviews. Other works focus on preservation or mitigation of specific properties, notably constant-time [CSJ+19, BGLP21].

Responsible disclosure. We informed the maintainers of libsodium and OpenSSL about our findings presented in Section 4.2.3. Both replied within a day acknowledging our findings and stating that they do not consider the findings a vulnerability, because neither library makes any claims about security in the attacker model we consider in this chapter. Consequently they also did not request an embargo. OpenSSL considers the finding a *"real security problem (at least on some platforms/compilers/architectures)"* in the form of *"a missing security hardening"*, for which they would be *"likely to accept a patch"*.

4.2 Failure Modes

One might think that overwriting used stack space and erasing values in registers before returning from a cryptographic routine should not be a difficult thing to do. Unfortunately this is not the case in many programming languages. In this section we look into different failure modes when attempting to protect cryptographic software in the attacker model we introduced in Section 4.1. We investigate approaches for stack zeroization taken by 9 popular open-source

crypto libraries written in C/C++ (possibly with assembly), namely BearSSL 0.6 [Por23], GnuTLS 3.6.12 [Gnu23], libsodium 1.0.18 [Lib23], mbedtls 3.3 [mbe], NSS 3.91 [NSS23], OpenSSL 3.1.1 [Ope23], TinyDTLS 0.9 [Tin23], and WolfSSL 5.6.3 [Wol23]. We also consider two libraries written in Rust, namely ring 0.17 [Bri23] and Dalek 2.0 (RC3) [Dal23].

While many of these libraries make an attempt to zeroize sensitive data, none of them explicitly claims security in the attacker model we use in this chapter. It should thus not surprise that none of the libraries is actually secure in this attacker model. In his reply to our disclosure e-mail, Denis, maintainer of libsodium, describes their approach as follows:

> *"Overwriting all secrets after use is not a goal of libsodium. It's silently done in a couple places where it's simple and cheap to do, but this is not a guarantee documented anywhere, and not something that's planned to be become one."*

In the subsequent subsections, we build up a hierarchy of failure modes and explain why languages like C/C++ or Rust and existing mainstream compilers make it close to impossible to not hit at least one of them.

4.2.1 Perform No Zeroization

A common technique of implementers is to ignore or shift the problem of memory scrubbing. The BearSSL crypto library, for example, recommends to overwrite the stack with garbage data after BearSSL was used. This shifts the responsibility of memory scrubbing to the application developer employing BearSSL. Unfortunately, this is not at all apparent to users of the library. Besides, even if an application developer was aware of their responsibility to clear the stack, they would probably fail to do so. This is simply because on a source code level it is completely unclear how much stack space was used by BearSSL.

The popular *ring* implementation of cryptographic primitives employs the same approach, and does not wipe sensitive data from memory either. Even though it is written in the memory-safe language Rust, it can be employed by C/C++ programs. A memory-corruption in the C/C++ application using *ring* can therefore also leak sensitive data previously processed within the library. This shows that writing cryptography in a memory-safe language does not eliminate the need for memory scrubbing.

4.2.2 Zeroization Falling Prey to Compiler Optimizations

A straightforward way to overwrite memory in C/C++ programs is to overwrite the desired memory region in a loop or by calling the `memset` function. Both approaches are inherently flawed when it comes to zeroization of sensitive data. The reason is that compilers will commonly apply an optimization called dead-store elimination (DSE), i.e., removing stores of variables that will not be read

anymore during their life time. Since zeroization occurs precisely once sensitive data is no longer needed, it is a textbook target for DSE.

From the libraries we investigated, TinyDTLS and NSS contain examples of attempts to zeroize sensitive data using `memset`. TinyDTLS calls `memset` directly to scrub memory from sensitive data, such as key material. NSS calls the macro `PORT_Memset`, which expands to `memset` on all platforms and, as demonstrated by [YJO+17], is eliminated by common compilers.

4.2.3 Zeroization in API Functions Only

Libraries that perform zeroization and put effort into avoiding the DSE pitfall typically use volatile function pointers to `memset` (OpenSSL), declare volatile memory regions (WolfSSL), or employ memory barriers (Libsodium). See also the detailed discussion in [YJO+17]. A more unified approach for zeroization in C is in principle offered by the `memset_s` function, which is guaranteed to not be eliminated by the compiler. However, since `memset_s` is part of the optional C17 appendix K [Int17], it does not have to be supported by a standard-compliant compiler. At the time of writing, no mainstream compiler supports `memset_s`.

Unfortunately, employing a zeroization routine that does not fall prey to DSE is by itself not sufficient to erase all sensitive data. All remaining libraries we investigated (i.e., those not listed in Sections 4.2.1 and 4.2.2) applied zeroization to sensitive data only selectively; typically only to secret data in the stack frame of API functions. Sensitive data derived from this secret data and contained in stack frames further down the call stack are not erased. An example of this scenario can be found in the ChaCha20 implementation of libsodium (and very similar in the ChaCha20 implementation of OpenSSL). The body of libsodium's ChaCha20 API function looks like this:

```
1  chacha_keysetup(&ctx, k);
2  chacha_ivsetup(&ctx, n, NULL);
3  memset(c, 0, clen);
4  chacha20_encrypt_bytes(&ctx, c, c, clen);
5  sodium_memzero(&ctx, sizeof ctx);
```

We can see that after the call to `chacha20_encrypt_bytes`, the local variable `ctx` containing the key is erased using the (carefully implemented) `sodium_memzero` routine. However, inside `chacha20_encrypt_bytes`, the entire context, including the encryption key, is copied onto the stack:

```
1  ...
2  uint32_t j0, j1,  j2,  j3,  j4,  j5,  j6,  j7, \
3           j8, j9, j10, j11, j12, j13, j14, j15;
4  ...
5  j0  = ctx->input[0];
6  j1  = ctx->input[1];
7  ...
8  j15 = ctx->input[15];
```

So even though the developer was actively zeroizing sensitive data in memory, the stack still contains a full copy of the secret key after libsodium returns control to the caller. To show that this is indeed an exploitable behavior, we developed a small example program with a memory-corruption bug that employs libsodium and leaks secret keys. The implementation of ChaCha20 in OpenSSL suffers from the exact same behavior.

The difficulty developers face by trying to identify all variables containing sensitive data is best described by the following commit message of Brian Smith, maintainer of the ring library [Bri23].

> *"Apart from that, by inspection, it is clear that there are many places in the code that don't call OPENSSL_cleanse where they "should". It would be difficult to find all the places where a call to OPENSSL_cleanse "should" be inserted. It is unlikely we'll ever get it right. Actually, it's basically impossible to get it right using this coding pattern."*

4.2.4 Zeroization on Source Level

Given the failure modes described in the previous subsections, one might think that zeroizing (potentially) sensitive data at the end of *all* functions, not just API functions, or zeroizing whenever a variable goes out of scope, might result in a sound solution. In Rust, there exist crates to implement this approach; most notably, the `zeroize` [Rus23b] and the `clear_on_drop` [Bar23a] crates. The former one being used, e.g., by Dalek [Dal23].

Unfortunately, even this approach is insufficient when implemented on source (e.g., C, C++, or Rust) level. The reason is that not all data that is placed on the stack by the compiler is visible on source level. The most obvious example for data that is written on the stack and that is not visible on source level are callee-saved registers that are spilled to the stack at the beginning of a function and restored to their original values before returning. More generally, the compiler is free to spill temporary variables on the stack as part of optimization and as a consequence it is largely out of the programmer's control what sensitive data is actually stored on the stack. For example, consider the `crypto_scalarmult` routine of the "`ref10`" implementation of X25519 [Ber06a] included in the SUPERCOP benchmarking framework and shown in Listing 1. The source-visible variables declared at the beginning of the function account for a total of 328 bytes when compiled for AMD64 (7 variables of type `fe` account for a total of $7 \cdot 40 = 280$ bytes, the array `e` uses 32 bytes, and the 4 `int` variables take a total of 16 bytes). However, compiling this code with GCC 10.2 and flags `-O3 -fstack-usage` computes a (worst-case) stack usage of 464 bytes. Some of the extra 136 bytes are simply padding for alignment, but most are used for source-level invisible data stored on the stack. Note that the `-fstack-usage` flag computes worst-case stack usage on a per-function granularity – it does not take into account stack space used by the functions *called by* `crypto_scalarmult`. See also Section 4.3.

```
typedef crypto_int32 fe[10];

int crypto_scalarmult(unsigned char *q,
                      const unsigned char *n,
                      const unsigned char *p)
{
  unsigned char e[32];
  fe x1, x2, z2, x3, z3, tmp0, tmp1;
  int pos;
  unsigned int i, swap, b;

  for (i = 0;i < 32;++i) e[i] = n[i];
  e[0] &= 248;
  e[31] &= 127;
  e[31] |= 64;
  fe_frombytes(x1,p);
  fe_1(x2);
  fe_0(z2);
  fe_copy(x3,x1);
  fe_1(z3);

  swap = 0;
  for (pos = 254;pos >= 0;--pos) {
    b = e[pos / 8] >> (pos & 7);
    b &= 1;
    swap ^= b;
[...]
  }
  fe_cswap(x2,x3,swap);
  fe_cswap(z2,z3,swap);

  fe_invert(z2,z2);
  fe_mul(x2,x2,z2);
  fe_tobytes(q,x2);
  return 0;
}
```

Listing 1: Source code of `crypto_scalarmult` from the `ref10` implementation of X25519.

Like others before us [Per14, SCA18, YJO+17] we conclude that it is impossible to protect against the attacker we consider in this chapter by zeroizing variables inside cryptographic routines on source level in commonly used languages such as C/C++ or Rust. It is thus not surprising that while many crypto libraries include some "best effort" stack zeroization, no library makes any claims about protecting against attackers who obtain residual stack data or content of registers after a crypto routine returns.

4.3 Possible Solutions

As it is impossible to offer a principled solution to zeroization *inside* the crypto routine *on source level*, we have two options: we either perform zeroization outside the crypto routine, i.e., on the caller side, or we work on lower than source level. We discuss these two options in the following.

4.3.1 Caller-Side Zeroization

Popular libraries such as BearSSL [Por23] recommend overwriting the stack after the call to API functions and offer routines to perform such zeroization on the caller side. The main limitation of this approach is that the caller has no information about how much stack space has been used by the crypto routine and thus needs to estimate stack usage. This is problematic, as a too low estimate could lead to secret data remaining on the stack and a too high estimate could violate stack size limits or even overwrite data in other segments, especially in an embedded setting. It might in principle be possible to compute the worst-case stack usage through per-function static analysis combined with call-graph analysis and forward that information from the build system. Both GCC and LLVM provide support for stack-usage analysis [GCC23, Rus23a], but we are not aware of any library using these features in their build system for zeroization. Furthermore, this approach does not take care of zeroing registers.

An elegant approach in a similar spirit is to allocate a new memory segment prior to entering a cryptography library's code. This segment is then used as stack space and completely wiped once the library returns control to the program. An implementation of this approach in Rust is provided by the `Eraser` crate [Spr22]. However, the caller would again have to make a guess about how much stack space is used in the library. Additionally, allocating a new memory segment and moving the stack to that segment requires platform and operating-system dependent subroutines. These low-level subroutines need to be written in platform-specific assembly to directly manipulate the stack-pointer register.

4.3.2 Callee-Side Zeroization

If we want to implement zeroization on the callee side, i.e., "clean up before we return", we need to either implement the complete crypto routine, including

zeroization, in assembly, or we require compiler support for zeroization of stack and registers. The first approach comes with the usual limitations of writing code in assembly: code is hard to maintain and error-prone, in particular when it comes to features like zeroization that are not covered by functional testing.

Consequently, the only remaining option for systematic zeroization is implementation in the compiler. Unsurprisingly, zeroization passes in mainstream compilers have been proposed before. GCC contributors discussed including a `clear_stack` function attribute for certain eligible functions and a `security_sensitive` attribute for variables [Gut16]. Similar work has been proposed for the LLVM compiler backend by Simon, Chisnall, and Anderson [SCA18]. Their work introduces a `__zero_on_return` function attribute that instructs the clang compiler to add a stack and register zeroization routine before a function's exit. In contrast to GCCs `clear_stack` proposal, this `__zero_on_return` approach also has an implementation employing the library's call graph. One thing these approaches have in common is that they still leave the developer in charge of deciding where zeroization happens by annotating what functions are "sensitive". More importantly though, these proposals for stack zeroization have unfortunately never been adopted in the mainline compilers and are thus not widely available to developers. Both GCC and clang however do support zeroing registers on function return with the `zero-call-used-reg` compiler option. This is possible as zeroing registers is a much simpler problem than zeroing the dynamically growing stack.

4.4 A principled solution in the compiler

In this section, we describe our principled approach for zeroing the stack. Our approach is integrated in the Jasmin compiler, and comes with formal guarantees of correctness and security. For completeness, we start by providing background on the Jasmin language.

4.4.1 Background on Jasmin

Jasmin [ABB+17, ABB+20a] is a programming framework for developing efficient, high-assurance cryptography. The framework is built around the Jasmin programming language, which lets programmers write efficient and readable code using "assembly in the head". Informally, Jasmin is an assembly-like language with structured control flow; in other words, it provides explicit access to assembly instructions (except `GOTO`s) for the architecture the developer is targeting, and has the usual control flow constructs **if**, **while** and function calls, as well as unrolled loops (denoted by **for**). Jasmin also supports zero-cost abstractions that increase readability and are predictably translated to assembly. One example of zero-cost abstraction is variables. A variable in a Jasmin program is an n-bit word that has either the reg or stack storage class, determining its storage (variables are not spilled by the compiler). One key benefit of the

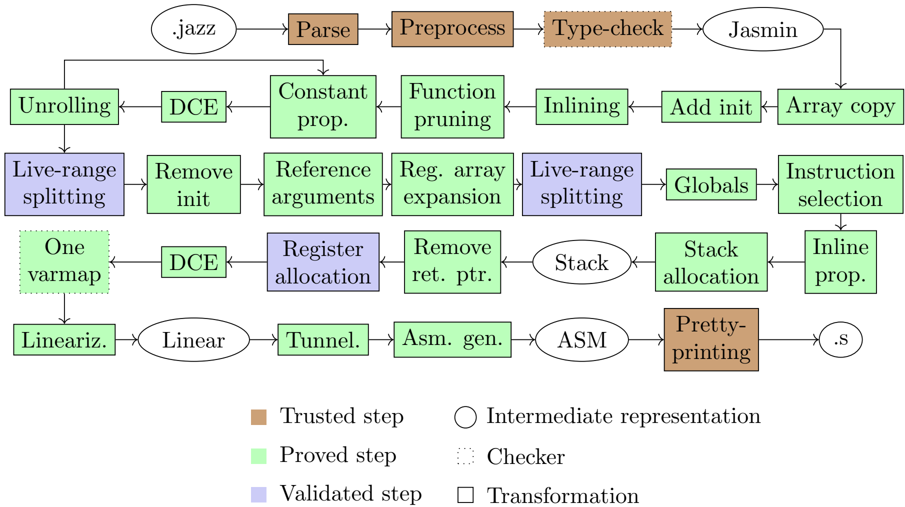


Figure 4.1: Program transformations in the Jasmin compiler

Jasmin language is that it comes with a machine-checked semantics that specifies the behavior of Jasmin programs; the semantics are written in the Coq proof assistant.

Jasmin programs are compiled using the Jasmin compiler. Currently, the compiler targets the AMD64 (aka, x86-64) architecture and has experimental support for ARMv7-M. Results are therefore also highly relevant for the embedded realm. The compiler consists of around 30 passes, summarized in Figure 4.1. We now describe the most relevant passes for our purposes. The initial passes are Jasmin to Jasmin transformations. For instance, the Inlining pass removes inline functions by inserting their bodies at their call sites. The Unrolling pass removes for loops by unrolling them. The Lowering pass replaces high-level assignments with specific assembly instructions, e.g. `x += y` with `ADD x, y` in x86-64. The Stack Allocation pass replaces stack variables by memory accesses relative to the stack pointer; it is in this pass that we can compute the amount of stack needed for the program. The Register Allocation pass assigns variables to architectural registers. The Linearization pass replaces structured control flow with `GOTO`s and labels, translating the program into a language called Linear. We insert our new stack and register zeroization passes after Linearization and before Tunneling, Assembly Generation and Pretty-printing.

To the exception of the parsing, preprocessing, type-checking and pretty-printing, each pass of the Jasmin compiler is certified in the Coq proof assistant. Following common practice in certified compilation, each pass is either implemented and verified in the Coq proof assistant, or it is implemented in an external language (OCaml) and its results are verified using a verifier which is implemented and verified in the Coq proof assistant. By combining the cor-

rectness results of each individual pass, one can prove that the Jasmin compiler is correct, i.e., it preserves the behavior of (safe) programs. This entails that functional correctness and reductionist security carry from source programs to the generated assembly.

In addition, the Jasmin framework provides a back-end to the EasyCrypt proof assistant [BGHZ11, BDG+14], which can be used for proving functional correctness and reductionist security. Last, the Jasmin framework provides guarantees about side-channel protections. In particular, there exist type systems for proving that Jasmin source programs are constant-time—the gold standard for protection against cache-based timing attacks in absence of speculation—or speculative constant-time—an enhancement of constant-time to protect against Spectre v1 attacks [KHF+19]. For details see [SBG+22].

The Jasmin framework has been used to develop efficient implementations of key cryptographic routines. In particular, Jasmin is the main medium used by the Libjade library, which provides formally verified implementations of post-quantum cryptographic algorithms [For23].

4.4.2 Design Choices

Any sound approach to zeroization must make four design choices: when should it be applied, what precision should it achieve, how should it be implemented, and where in the compilation chain should it be performed? We comment on each design choice in turn.

When to zeroize? The first question is when should zeroization be applied. In order to achieve the best trade-offs between security and performance, the ideal compromise is to apply zeroization at the interface between cryptographic routines and application-level code. Conveniently, Jasmin offers a clean interface between cryptographic code and the caller. Specifically, Jasmin programs contain **export** functions and internal functions. Internal functions cannot be called from the outside. On the contrary, **export** functions are the entry point of the programs, and the exit of the **export** function is the point where we exit the Jasmin world and return to the outside world. This is a natural place to introduce the zeroization code. However, this does not mean that we do not clear the memory used by the internal functions! At the end of the **export** function, the code injected by the compiler zeroizes the entire stack used during the call of that function, that is, not only the stack frame of the **export** function, but also the stack space used by the internal functions it transitively calls.

What precisely to zeroize? Any approach to zeroization must decide between precision and simplicity. On the one hand, minimizing the number of memory writes could potentially minimize overhead. On the other hand, a complex approach may be more difficult to implement, even more so because the Jasmin compiler is verified, without necessarily bringing any significant performance improvement over a naive approach.

We choose to implement a simple and systematic approach, where the injected code blindly clears all used stack space and auxiliary registers, with the hope that the zeroization will be cheap anyway. This hope was also motivated by the fact that stack allocation in the Jasmin carefully minimizes overall stack usage by re-using space when stack arrays go out of scope. The hope was confirmed by our benchmarks, see Section 4.5.

How to implement zeroization? The natural implementation of stack zeroization is using a loop, repeatedly clearing the stack one chunk at a time. But since the bound on the stack size is known at compile time, we can also unroll this loop and generate sequential code performing zeroization. The first approach has the advantage of keeping the code size overhead small, the second one should typically be faster. We can also imagine mixing both approaches, i.e., use a partially unrolled loop. However, we choose to implement only these two strategies, that we call *loop* and *unrolled*, respectively.

Another design choice is the clearing instruction that is used at each step. We simply use `MOV` and `VMOV` to write zeros. As for what size to clear at each step, here again we choose the simple approach. All the instructions introduced clear the same number of bits. This implies that we can clear only a size that is a multiple of this number. If the stack size is not a multiple, we round up and clear the next largest multiple of stack bytes. This means that enabling the zeroization has an impact on the analysis mentioned in the previous paragraph. Details are given in the **Analysis** paragraph in Section 4.4.3. The user can select the size cleared at each step, either 8, 16, 32, 64, 128 or 256 bits. If this is less than or equal to 64 bits, we use `MOV`. Otherwise, we use `VMOV`. There is also a default value which is the stack alignment of the **export** function.

Where in the compiler should zeroization be performed? The insertion of the zeroization pass in the compiler chain involves several considerations. First, zeroization can only be performed once the compiler has fixed the stack space used by the **export** function and all functions further down the call stack. In the Jasmin compiler, this is done during Stack Allocation, so we must insert the Zeroization pass after Stack Allocation. We choose to perform Zeroization after Linearization, although it could go at any point after Register Allocation.

4.4.3 Implementation Overview

Implementing zeroization in the compiler comprises two main tasks: computing what to clear (which regions of memory need zeroizing) and generating the corresponding assembly instructions.

Analysis. During the Stack Allocation pass, the compiler computes two pieces of information for each function (whether it is **export** or internal): the size of the stack frame used and the minimal stack alignment required. For the

frame size, the compiler determines a frame layout, trying to share stack memory for local variables (meaning that if a stack variable is dead, then it tries to reuse that stack region), and then it deduces the amount of stack needed by the function. The stack alignment is deduced from the writes to the stack. It is the minimum alignment required so that all writes to the stack are aligned.

The compiler does this for each function, and then propagates the information to account for function calls, from the callees to the callers, summing the stack frame sizes and taking the maximal alignment. Each function must indeed take into account the stack used by the functions it calls, directly or transitively. Since Jasmin does not support recursive functions, the call graph is acyclic and this propagation is not difficult. The result is an upper bound on the amount of stack used and an alignment for each function, taking function calls into account. It is an upper rather than an exact bound because, for instance, the function might call some other function conditionally with respect to a runtime value, so the analysis must be conservative. This bound is tight in the sense that there exists an execution that uses this amount of stack. The same reasoning with respect to conditional calls applies to the alignment.

When stack zeroization is applied, the analysis is amended. Three changes are made. First, if the size to be cleared at each step is greater than the alignment of the **export** function, the alignment is increased so that both match. This is needed so that the writes introduced by the zeroization are aligned. Second, the frame size of the **export** function is rounded up, so that it is a multiple of the alignment, meaning that some padding is inserted in the frame. To explain that point, we must give some details about the code generated by the Jasmin compiler for **export** functions. To allocate the frame of the **export** function on the stack, the stack pointer is first decreased by the size of the stack frame, then aligned on the alignment of the function. In the clearing code, we start by aligning the stack pointer. Roughly speaking, we need the decrease and the alignment to commute. One way to achieve that is having the frame size be a multiple of the alignment. Third, the bound on the stack size is rounded up, so that it is a multiple of the size of the clear step. This is to take into account that, as mentioned in Section 4.4.2, we can clear only a size that is a multiple of the size of the clear step.

Transformation. The compiler injects zeroization code at the end of **export** functions. At that point, the stack pointer is already restored to its initial value. For that reason, the zeroization code has to redo the alignment performed at the entry of **export** functions. It copies the stack pointer to another register and computes the alignment. Then it introduces a loop or a sequential code, depending whether the strategy is loop or unrolled.

Interestingly, some modification to the compiler was required for implementing the transformation. In particular, the formal development was modified to support compilation passes which introduce new labels in the Linear program.

```
1  fn f () ⟶ reg u64
{
2      stack u8 s;
3      ...
4      return r;
5  }
6
7  export fn main ()
⟶ reg u64 {
8      stack u32 s;
9      ...
10     r = f ();
11     return r;
12 }
```

(a) Jasmin program

```
main:
  movq %rsp, %rsi          // Save the SP.
  leaq -8(%rsp), %rsp      // Allocate a word.
  andq $-8, %rsp           // Align.
  ...
  ...                      // Code.
  ...
  movq %rsi, %rsp          // Restore SP.
  andq $-8, %rsp           // Align.
  subq $16, %rsp           // Point at max.
  movq $16, %rdi           // Set up counter.
zloop:
  subq $8, %rdi
  movq $0, (%rsp,%rdi)     // Zeroize.
  jne zloop
  movq %rsi, %rsp          // Restore SP.
  ret
```

(b) Code produced with loop strategy

```
main:
  movq %rsp, %rsi          // Save the SP.
  leaq -8(%rsp), %rsp      // Allocate a word.
  andq $-8, %rsp           // Align.
  ...
  ...                      // Code.
  ...
  movq %rsi, %rsp          // Restore SP.
  andq $-8, %rsp           // Align.
  subq $16, %rsp           // Point at max.
  movq $0, 8(%rsp)         // Zeroize
  movq $0, (%rsp)          // Zeroize
  movq %rsi, %rsp          // Restore SP.
  ret
```

(c) Code produced with unroll strategy

Listing 2: A schematic Jasmin program **(a)**, and the assembly produced with a clear step of 64 bits, for strategies loop **(b)** and unrolled **(c)**.

Illustrative example. Let us consider the schematic Jasmin program shown in Listing 2, on the left and let us first illustrate the analysis on it. Function **f** uses 1 byte of stack, but actually it also has to save the return address on the stack, which requires 8 bytes, so 9 bytes in total. The alignment is 64 bits, because of the 64-bit return address. For alignment reasons, the frame size of internal functions are rounded up to a multiple of the alignment, so the final frame size is 16 bytes. Function **main** uses 4 bytes of stack for itself, and calls **f**. This results in 20 bytes being used by **main**. As for the alignment, **main** inherits the alignment of **f**, 64 bits.

Let us assume that the user applies the stack zeroization feature with the default clear step size, 64 bits, corresponding to the alignment of **main**. The alignment of **main** is unchanged, but the frame size is rounded up and becomes 8 bytes. **f** still uses 16 bytes of stack. The stack size used by main is thus 24 bytes. This is already a multiple of the clear step size, 64 bits, so it does not need to be rounded up.

The resulting assembly for strategies loop and unrolled is shown in figure 2, in the middle and on the right, respectively. We can observe that in both cases, the clearing code start with aligned the stack pointer, mimicking the initial alignment at the start of **main**. In the loop strategy, a counter is set up before entering the loop. The counter is decreasing, the loop exits when it reaches 0. Written this way, it is enough to check the flags set by `addq`, there is no need to call a comparison operator. In the unrolled case, the code consists in as many clearing instructions as needed. In this example, we need 3.

4.4.4 Register and Flag Zeroization

The compiler also offers to developers the possibility to zeroize registers, flags or XMM registers, or a combination of these. No analysis is needed in this case, since the targets of zeroization are known.

4.4.5 Correctness and Security

We have proved that the Jasmin compiler with the zeroization pass achieves correctness and security. Both statements are stated as end-to-end results, i.e., as relations between source code and assembly code. The (simplified) statements of correctness and security are displayed in Figure 4.2.

The correctness result is a classic simulation, stating that if an **export** function f_s from the source program transforms state σ_s into state σ'_s, and its compilation f_t in the compiled program transforms σ_t into σ'_t, and if σ_s is equivalent to σ_t, then σ'_s is equivalent to σ'_t, where $\equiv$ is a relation between source states and target states. End-to-end correctness of the compiler follows from correctness of each individual pass. The correctness statement of other passes could be reused.

The security result states that under the assumptions above, all memory locations in σ'_t whose addresses are undefined in σ_s either retain their value

$$\begin{array}{ccc} \sigma_s & \xrightarrow{f_s} & \sigma'_s \\ \equiv \vdots & & \\ \sigma_t & \xrightarrow[f_t]{} & \sigma'_t \end{array}$$

- Correctness: $\sigma'_s \equiv \sigma'_t$
- Security: $\neg\,\mathsf{valid}(\sigma_s, p) \implies \sigma'_t\,[p] = \sigma_t\,[p] \lor \sigma'_t\,[p] = 0$

where $\equiv$ is a relation between source and assembly states and $\neg\,\mathsf{valid}(\sigma_s, p)$ says that p is not a valid address w.r.t. source state σ_s.

Figure 4.2: Correctness and security of zeroization.

from σ_t or contain zeros (see Figure 4.2). The compiler only writes stack data to memory addresses that are invalid in σ_s, so it follows that everything an adversary learns about the stack from σ'_t was present in σ_t already. In fact, our security statement considers a more refined attacker model than previously discussed: when an **export** function returns, the attacker acquires the memory contents between sp and $sp - n$, together with registers and flags, where sp is the stack pointer before the function executed and n the amount of extra stack memory needed by the compiler. Our transformation ensures that all the memory given to the adversary is either valid and originates from the client code, or is equal to 0. Note that in this model the attacker learns the amount of used stack, but this is public since it's statically determined from the program.

The proof of security is done for the zeroization pass. Then, one proves that the security property is preserved by all subsequent passes. However, in order to establish our end-to-end statement, it has also been necessary to prove that passes before zeroization do not introduce arbitrary writes. For instance, we have proved that stack allocation respects stack usage predicted by the compiler, i.e. it does not write outside of the region of the stack that the compiler predicted to use. The zeroization property for the entire compiler is as follows: If an **export** function f_s from the source program transforms state σ_s into state σ'_s, and its compilation f_t in the compiled program transforms σ_t into σ'_t, and if σ_s is equivalent to σ_t, then memory locations in σ'_t whose addresses are undefined in σ_s either retain their value from σ_t or contain zeros.

4.4.6 Combining Leakage Models

As stated in the introduction, our threat model so far is limited to an attacker that can observe the contents of the registers and flags upon return of the cryptographic routine, as well as the contents of the stack used by the cryptographic routine. This threat model is orthogonal to threat models for micro-architectural attacks, including cache attacks, or speculative attacks like

Spectre. In this section, we sketch how our countermeasure can soundly be combined with existing approaches to protect against this type of side-channel attacks. For completeness, we briefly review the prominent models for cache-based timing attacks, without and with speculative execution.

The constant-time policy. In absence of speculative execution, the baseline for side-channel protection is (cryptographic) constant-time. Informally, the constant-time policy considers a leakage model where program execution leaks all control-flow decisions and all addresses (not values) of memory accesses. The constant-time property states that an attacker cannot learn any secret information from leakage. Formally, the property is stated relative to a notion of indistinguishability between states, where two states are indistinguishable if they coincide on the fragment of the memory that can be accessed directly by the attacker. Then, we say that a program is constant-time if leakage cannot separate between any two executions starting from indistinguishable states.

We can now combine the two threat models, and consider an attacker that can observe both the constant-time leakage and the contents of the stack, registers and flags after program execution does not learn anything about secrets. For lack of a classic name, we call this model stack constant-time. It is not too hard to observe that zeroization transforms a constant-time program into a stack constant-time program. Informally, this is a consequence of the correctness of stack zeroing, and of the following two observations. First, the part of the stack that is zeroed by zeroization is determined statically and independently of the values held in memory. Second, zeroing the same part of the stack preserves state indistinguishability.

The speculative constant-time policy. The baseline for side-channel protection in presence of speculative execution is speculative constant-time. The leakage model is similar to the one of constant-time. However, the attacker can now actively influence all control-flow decisions, and the addresses of unsafe memory reads and writes carried during misspeculation. More formally, the operational semantics of programs is extended with a set of directives that are controlled by the attacker and determine the control-flow of the program. Then, a program is speculative constant-time if, for every choice of the attacker at control-flow points, leakage cannot separate between any two speculative executions starting from indistinguishable states.

Similar to the previous case, we can define a notion of stack speculative constant-time. However, in this case one cannot prove that our zeroization procedure using loops transforms a speculative constant-time program into a stack speculative constant-time program. This is because the transformation does not prevent early abort attacks, and so an attacker with control over the branch predictor can force the whole zeroization to be skipped [SBB+22]. To avoid this attack, we offer a compiler flag to add a fence at the end of the zeroization loop. Note that the attack does not work on the variant of zeroization using an unrolled loop. So, in both cases, we can prove that zeroiza-

tion transforms a speculative constant-time program into a stack speculative constant-time program.

Integration with the Jasmin compiler. Note that in contrast with the correctness and security results of the previous section, the claims of this paragraph are not machine-checked in the proof assistant, and they are not limited to the zeroization pass. Formalizing the results in the Coq proof assistant would be possible with reasonable effort. However, there are some main obstacles to extend them to end-to-end results. We discuss these obstacles below.

In the case of constant-time, the end-to-end result would state that the Jasmin compiler with zeroization transforms a constant-time source program into a stack constant-time program. A prerequisite for proving this end-to-end result would be to prove that the Jasmin compiler preserves constant-time. Indeed, such a preservation result exists. Unfortunately, it has been proved for a previous version of the Jasmin compiler [BGLP21] and the proof has not yet been merged into the latest state in the `main` branch.

In the case of speculative constant-time, there is currently no formal guarantee that the compiler preserves speculative constant-time (for Spectre v1). This remains an exciting direction for future research.

4.5 Benchmarks and Validation

This section evaluates the cost of zeroization. As a starting point, we use the artifact from [SBG+22], which protects cryptographic implementations from Libjade [For23] against Spectre v1. We produced the data in this section with a machine with Linux Debian 5.10.0-21-amd64, GCC 10.2.1, and equipped with an Intel Core i7-10700K (Comet Lake) with hyperthreading and TurboBoost disabled.

Table 4.1 reports the cycle counts for six cryptographic primitives and nine implementations. For implementations with variable-input-length, such as ChaCha20 and Poly1305, we include the measurements for 128, 1024, and 16384 bytes. Each reported value in Table 4.1 corresponds to a median of 10000 executions. Given that the overhead of zeroization from our approach is small (from an absolute perspective), we repeated the experiment a total of eleven times, and included in the table the median of these (which can be interpreted as an approximation of the median of 110000 executions).

The fourth column from Table 4.1, **Baseline**, corresponds to the results of our experiments without performing any zeroization. The fifth column, **Loop**, presents the cycle counts when we perform a complete stack zeroization using a loop with a fence instruction after it, to prevent early-abort speculative execution attacks (`-stack-zeroization loopSCT`). In addition to the stack cleaning, for reference implementations (ref in the table) we clear all 64-bit registers that might leak, and for vectorized implementations, we also set the 256-bit registers to zero. The overhead of the **Loop** setup (compared to the

Baseline) is shown in the next column. The highest overhead in this column is 32.56%, corresponding to the vectorized version of Poly1305 when operating on 128 bytes of input data: the cost of zeroization is fixed, and for this particular case, the average expected overhead is just 56 CPU cycles. This cost is quickly amortized for larger inputs and converges to overheads close to zero, demonstrated by the 1.09% overhead for the same implementation and an input length of 16 KiB.

The CPU cycles reported in column **Unroll** correspond to a straight-line code zeroization (option `-stack-zeroization unrolled`). The register zeroing is performed in the same way as in **Loop**. Generally, the overhead of zeroing the stack using the straight-line code is lower when compared to zeroization using a loop. Both overhead columns show a negative value for the scalar multiplication of X25519, -0.02% and -0.04%, for the loop and unrolled variants, respectively. We suspect this result to be due to different code alignment resulting from zeroization and will continue to investigate the matter. The computation overhead for zeroing the state in all Kyber's operations is small and below 2% for all reported measurements except one, Kyber512 decapsulation. Kyber768 avx2 implementation uses 15392, 18432, and 19552 bytes of the stack in keypair, enc, and dec, respectively, and if we consider that cost of clearing the stack lies between 400 and 600 CPU cycles, on average, roughly 32 bytes of stack are cleared each CPU cycle.

In the context of code size, the assembly file produced by the Jasmin compiler for Kyber768 avx2 has 33761 lines for the **Baseline** version. For the **Loop** and **Unroll** variants, this increases to 33869 and 35519, respectively. In **Unroll**, one instruction is issued for every 32 stack bytes: in the particular case of dec, which uses 19552, this corresponds to 611 instructions.

Overall, Table 4.1 shows that the overhead is very small and in many cases barely measurable for both variants, so in scenarios where code size is a concern and stack space usage is intensive, using the **Loop** option is recommended; otherwise, the **Unroll** option performs better on average.

Validation. To validate the effectiveness of stack zeroization code, we call each function from the Libjade API in a C wrapper that reads the region of memory that the function used as stack. More specifically, we implement a test case for each library function, where we first fill the memory region that the function will use as stack with the output of a PRF. Then, we call the Libjade function and subsequently assert that the memory region we filled with the PRF was zeroized (in our current implementation this means overwritten with zeros). On the other hand, for register zeroization we inspected the assembly output of the compiler and ensured each register was being overwritten with zeros.

Table 4.1: Benchmark results on an Intel Core i7-10700K (Comet Lake) CPU including introduced overhead in percent.

Primitive	**Impl.**	**Op.**	**Baseline**	**Loop**	**% ov.**	**Unroll**	**% ov.**
ChaCha20	avx2	128 B	372	458	23.12	398	6.99
	ref	128 B	796	840	5.53	810	1.76
	avx2	1 KiB	1254	1304	3.99	1256	0.16
	ref	1 KiB	5996	6038	0.70	6008	0.20
	avx2	16 KiB	19076	19104	0.15	19144	0.36
	ref	16 KiB	94618	94646	0.03	94660	0.04
Poly1305	avx2	128 B	172	228	32.56	184	6.98
	ref	128 B	172	212	23.26	176	2.33
	avx2	1 KiB	684	744	8.77	702	2.63
	ref	1 KiB	1044	1080	3.45	1052	0.77
	avx2	16 KiB	8422	8514	1.09	8468	0.55
	ref	16 KiB	15932	15964	0.20	15970	0.24
secretbox	avx2	128 B	1244	1342	7.88	1282	3.05
	ref	128 B	1702	1760	3.41	1710	0.47
	avx2	1 KiB	3110	3216	3.41	3158	1.54
	ref	1 KiB	8006	8052	0.57	8028	0.27
	avx2	16 KiB	31342	31434	0.29	31450	0.34
	ref	16 KiB	115402	115426	0.02	115496	0.08
X25519	mulx	smult	98334	98432	0.10	98304	-0.03
Kyber512	avx2	keypair	25884	26256	1.44	26282	1.54
	avx2	enc	35416	35860	1.25	36024	1.72
	avx2	dec	27984	28886	3.22	28470	1.74
Kyber768	avx2	keypair	43096	43402	0.71	43352	0.59
	avx2	enc	55134	55490	0.65	55268	0.24
	avx2	dec	44294	44938	1.45	44756	1.04

4.6 Conclusion and Future Work

The solution to zeroization we presented in this chapter offers many desirable properties: it is principled in the sense that it is guaranteed *that* sensitive data on the stack and in registers is cleared and it is well defined *when* this happens. As we showed in Section 4.5, the cost for zeroization is typically very small. This is inherent to our solution and therefore also translates to other architectures, such as `Armv7-M`, used in many embedded settings and experimentally supported by the Jasmin compiler. Integration into the Jasmin compiler releases library developers from the error-prone task of taking care of zeroization and ensures that zeroization takes place on the callee side, where the responsibility naturally resides.

There are also some drawbacks to our solution, most obviously that it requires writing cryptography in the Jasmin programming language. This may not be an option for various reasons, most notably that, so far, the Jasmin compiler only targets the AMD64 (and, experimentally, ARMv7-M) architectures. However, for projects that *can* integrate assembly code generated by the Jasmin compiler, switching to Jasmin implementations comes with additional benefits like certified compilation, timing-attack and Spectre-v1 protection [SBG+22], and an interface to EasyCrypt [BGHZ11] proofs of functional correctness and reductionist proofs of security. One obvious line of future work is to extend Jasmin's set of supported target architectures, including the zeroization support we presented here.

Furthermore, recall that our proposed approach erases *all* data and does so only when returning from an **export** function. One can imagine investigating more fine-grained approaches in two different senses: First, as Jasmin features a information-flow type system, one could decide to erase secret data only. While the information about secrecy is in principle available, it is not trivial to decide at the end of an **export** function for each address in the used stack space, if the last write contained secret data or not. Not only would this approach add considerable complexity and in most cases offer only small benefits, it would also create a performance conflict with Spectre v1 protection: For the selective SLH countermeasures implemented in [SBG+22], it is beneficial to declare data as "secret" whenever this data is not used as address or branch condition, even if it not secret from a cryptographic point of view. This approach of declaring as much data as possible as "secret" would, for most cryptographic routines, mean that anyway almost all stack data needs to be cleared. Distinguishing "truly secret" and "additional secret" data in the type system would probably be possible, but again add complexity in both the compiler and Jasmin implementations. Second, one could decide to implement more frequent zeroization, for example at the end of every (also Jasmin-internal) function or even whenever a variable goes out of scope. Again, given on our benchmarks of fast functions on short inputs, we expect such an approach to be quite expensive in terms of performance.

The Jasmin compiler is an ideal context for developing security-aware com-

pilation techniques, for three reasons. First, Jasmin is primarily targeted to cryptographic software, which mandates the use of these transformations. Second, the Jasmin compiler is arguably simpler than a mainstream compiler, making the implementation and integration of these transformations much simpler. Last, the overarching goal of the Jasmin/EasyCrypt approach to high-assurance cryptography is to derive assembly-level guarantees against implementation adversaries. As noted by Percival, proving properties such as forward secrecy at assembly-level mandates the use of a formally verified zeroing compiler, so there is a very strong motivation to adopt and maintain zeroization in the Jasmin compiler. Nevertheless, we express our hope that eventually also mainstream compilers will offer a clean approach to memory zeroization, for example, by adopting the solution proposed in [SCA18] for LLVM.

Part II

Securing Protocols

Chapter 5

KEMTLS vs. Post-Quantum TLS in Resource-Constrained Devices

This chapter is built upon work from the following paper:

> Ruben Gonzalez and Thom Wiggers. KEMTLS vs. post-quantum TLS: Performance on embedded systems. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 99–117. Springer, 2022. `https://eprint.iacr.org/2022/1712`

In it, we conduct a performance comparison of the established TLS 1.3 and the newly proposed KEMTLS in an embedded setting, including different transmission mediums. The comparison has a focus on server authentication within the embedded client, but takes the full post-quantum handshake into account. TLS secures transport for high-end desktops and low-end embedded devices alike. However, especially for small microcontrollers, it appears the current NIST-selected post-quantum signature solutions pose a challenge. Dilithium e.g. suffers from very large public keys and signatures; while Falcon has significant hardware requirements for efficient implementations.

KEMTLS is a proposal for an alternative TLS handshake protocol that avoids authentication through signatures in the TLS handshake. Instead, it authenticates the peers through long-term KEM keys held in the certificates. The KEMs considered for standardization are more efficient in terms of computation and/or bandwidth than the post-quantum signature schemes.

To gain meaningful results, this chapter presents implementations of KEMTLS and TLS 1.3 on a Cortex-M4-based platform. These implementa-

tions are based on the popular WolfSSL embedded TLS library and hence share a majority of their code. In the experiments, both protocols are investigated with the NIST finalist signature schemes and KEMs, except for Classic McEliece which has too large public keys. Both protocols are benchmarked and compared in terms of run-time, memory usage, traffic volume and code size. The benchmarks are performed in network settings relevant to the Internet of Things, namely low-latency broadband, LTE-M and Narrowband IoT.

Since publication of the paper, two of the three investigated signature algorithms have been standardized with very slight modifications (Dilithium) or have been selected for standardization (Falcon). Results therefore remain as relevant as at the time of writing. The third investigated algorithm, Rainbow, however, has been eliminated from the standardization process as Beullens presented an efficient attack [Beu22]. Newly submitted multivariate signature schemes, such as MAYO or UOV, are still considered for standardization by NIST. Results based on the related Rainbow can still be of utility when extrapolated to those schemes. The KEM algorithms SABER and NTRU were not broken, but also did not get standardized by NIST. They can (and are) still be utilized by researchers and projects not bound to NIST standards though. Yet, for clarity, eliminated schemes are marked as such in the results.

Contribution. This work investigates if KEMTLS' advantages transfer to the embedded realm by comparing KEMTLS and TLS with PQC (PQTLS) in an embedded setting. For this purpose, KEMTLS and PQTLS were implemented including all NIST finalist signature schemes and KEMs, except for Classic McEliece. As the PQTLS and KEMTLS implementation share large parts of their code base, a direct performance comparison is possible. The analysis focuses on the relevant trade-offs embedded systems engineers face. To our knowledge, this is the first work to investigate KEMTLS in an embedded setting. The benchmarks include runtime, memory usage, code size and bandwidth consumption of our KEMTLS and PQTLS instantiations. Results were obtained by running the implementations on a Cortex-M4-based platform. Experiments were conducted with a technology stack that is typically used in real-world deployments, in which the embedded device is a TLS client talking to a TLS server running on a high-end computer. This computer also simulated different network technologies throughout the experiments.

Associated Software. The software developed for this chapter includes a forked version of the PQM4 library that offers WolfSSL compatible APIs, namely a *verify* function on the signatures and proper namespacing for using more than one primitive in a code base. It further contains an integration into the ZephyrOS embedded real time operating system and a Docker-based reproducible build system. All code needed to replicate the results is available at `http://doi.org/10.5281/zenodo.17381244`.

Organization of this chapter. After a brief introduction on previous work and progress of the standardization process in Section 5.1, the PQTLS and

KEMTLS protocols are presented in Section 5.2. To support results, in Section 5.3 the implementation and experimental setup will be explained in detail. Finally, the results will be presented in Section 5.4, discussed in Section 5.5 and concluded in Section 5.6.

5.1 Introduction

Transport Layer Security (TLS) is ubiquitous in modern computer networks. It adds confidentiality, authenticity and integrity to application-layer protocols. We trust it, among other things, with securing connections to websites, emails, instant messages and virtual private networks. In its most recent version, TLS 1.3 [Res18], ephemeral (elliptic curve) Diffie-Hellman is used to establish encryption keys. Server (and optionally client) authentication is achieved by using digital signatures. To verify the signatures, public keys are transmitted in certificates during the TLS handshake. These certificates are in turn signed by certificate authorities, which are pre-installed on the verifying device.

As TLS is an integral part of today's internet security architecture, it is vital to integrate post-quantum cryptography soon. This promises to mitigate the increasingly grave threat of large quantum computers to cryptography.

While the NIST standardization and industry adoption of PQC primitives is still ongoing, work on post-quantum TLS (PQTLS) has also begun. This began with academic experiments in 2015, demonstrating R-LWE key exchange in TLS 1.2 [BCNS15]. Like this previous work, others have mostly focused on the ephemeral key exchange in TLS, often using so-called "hybrid" algorithms. These essentially perform a classic elliptic-curve key and a post-quantum key exchange in parallel, to increase the confidence in the security. Google and Cloudflare have already conducted large-scale industry studies employing hybrid algorithms within TLS [KLS+19, Lan19, Lan18]. Amazon already includes experimental support for post-quantum schemes in its S2N TLS implementation and its Key Management Services product [Hop19]. While previous works have mainly focused on post-quantum confidentiality; there have been fewer experiments deploying post-quantum authentication. This is because "harvest now, decrypt later" attacks pose a more imminent threat to confidentiality than active attacks pose on authentication. Sikideris et al. [SKD20] have measured the performance of post-quantum signature schemes between servers in two data centers. They concluded that out of the (NIST Round 2) schemes they tested, only two (Falcon [PFH+20b] and Dilithium [BLD+20]) seem viable for deployment in TLS 1.3. Experiments by Cloudflare [Wes21], that added dummy data to TLS connections to measure the impact of the larger sizes of post-quantum signature schemes, seem to support these results. Still, when using Dilithium as a drop-in replacement for all of the signatures in TLS (which adds 17 kB to the handshake), Cloudflare reports an expected 60–80% slowdown for the Linux default congestion window of 10 packets. Falcon has much more favorable public key and signature sizes but requires hardware support

for double-precision floating-point operations. Without this, Sikideris et al. report signing handshakes with Falcon is not viable.

There has also been some work investigating embedded devices rather than large-scale, high-performance computers. Bürstinghaus-Steinbach et al. have experimented with Kyber and stateless hash-based signature scheme SPHINCS+. They integrated SPHINCS+ in mbedTLS's [mbe] TLS 1.2 implementation and showed the performance on various Arm boards [BSKNS20]. More recently, George et al. have evaluated the performance of post-quantum TLS 1.3 on embedded systems [GLF+21]. They investigated the performance of the NIST finalist KEMs and the Dilithium and Falcon signature algorithms in WolfSSL's TLS 1.3 implementation.

To mitigate the difficulties with post-quantum signatures, Wiggers et al. proposed KEMTLS [SSW20]. Instead of authenticating the handshake through a signature, KEMTLS performs authentication through KEM key exchange with KEM public keys in the certificates. As the KEMs currently considered for standardization are generally smaller and/or more computationally efficient than the post-quantum signature schemes, this can be more efficient. Additionally, it reduces the size of the trusted code base: the code that facilitates the ephemeral key exchange can also be used for authentication. Knowledge of the server's long-term key is imaginable in e.g. session resumption, or perhaps in IoT applications where the clients speak to a single server. We note that the certificates are still signed by a certificate authority using post-quantum signatures. We thus still need to verify post-quantum signatures; we might however choose signature algorithms that are optimized for size or verification time rather than signing time.

While KEMTLS and PQTLS have been compared, these studies focused on high-end hardware and high bandwidth connections [SSW20, SSW21, CFS+21]. However, TLS is used for more than just protecting web browsing on desktop computers. The Internet of Things (IoT) increasingly interconnects embedded devices over the internet. Especially device-to-cloud communication is an omnipresent IoT use case. New communication protocols like Matter [Con22] (formerly Connected Home over IP) mark a new trend by using IPv6 and forcing every embedded device to establish its own end-to-end-secure connection. From a security perspective, this makes perfect sense. However, for embedded software developers this poses a challenge. Key establishment, digital signatures and certificate transmission are already problematic for low-cost, resource-constrained devices. With the advent of post-quantum cryptography, it will become even more challenging to establish TLS connections from those embedded devices.

5.2 Background

In this section, we will give background on the impact post-quantum cryptography has on TLS 1.3 and the development of KEMTLS.

5.2.1 Post-Quantum TLS

We will briefly explain how TLS 1.3 post-quantum can be made post-quantum and summarize the KEMTLS proposal for an alternatively authenticated TLS handshake.

(Post-quantum) TLS

TLS is a protocol that has seen widespread deployment, famously as part of HTTPS. Its most recent iteration is TLS 1.3 [Res18]. In the most common uses, it offers unilateral authentication of the server to the client. Optionally, it also allows mutual, client-to-server authentication. The protocol authenticates the peers through signatures, which are in turn verified using public keys that are contained in (CA-signed) certificates. There is optional support for pre-shared, symmetric keys in place of certificate authentication as well.

The unilateral, certificate-authenticated TLS 1.3 handshake consists of an ephemeral, Diffie–Hellman (DH) key exchange followed by a signature over the handshake to authenticate. Finally, the handshake is additionally authenticated by a MAC. It is possible to make this handshake post-quantum, simply by replacing the server's DH key generation with $\mathsf{KEM_e.Encapsulate}$, to encapsulate against the client's ephemeral public key, and sending the ciphertext instead of the server's ephemeral DH public key. The client would derive the shared secret by decapsulating the ciphertext. For authentication, we simply use post-quantum signature algorithms in place of RSA or elliptic curve signatures. The TLS 1.3 handshake has been very carefully designed to be very efficient in the number of round-trips and is a single round-trip (1-RTT) protocol. As we can see on the left-hand side of Figure 5.1, the server can already send data to the client in its first response flow. The client can send it first message after the server's first flow, after 1.5 RTT.

KEMTLS

Post-quantum KEMs and signature algorithms differ in their characteristics more than their pre-quantum equivalents. KEMTLS is an alternative proposal for a PQTLS handshake, which allows using KEMs (which are typically much smaller and/or more computationally efficient than post-quantum signatures) instead of signatures in the online handshake. In KEMTLS, the certificates contain public keys for a KEM instead of a signature scheme. There are still signatures for the verification of the certificate chain, but these only need to be verified. As those signatures are done offline, it is also possible to use algorithms optimized for public key and signature size, rather than signing time. For example, [SSW20, Appendix D] gives parameters for such a variant of $\text{XMSS}^{\mathsf{MT}}$.

KEMTLS is inspired by the OPTLS proposal by Krawczyk and Wee [KW16]. OPTLS was an early proposal for TLS 1.3, where the authentication would

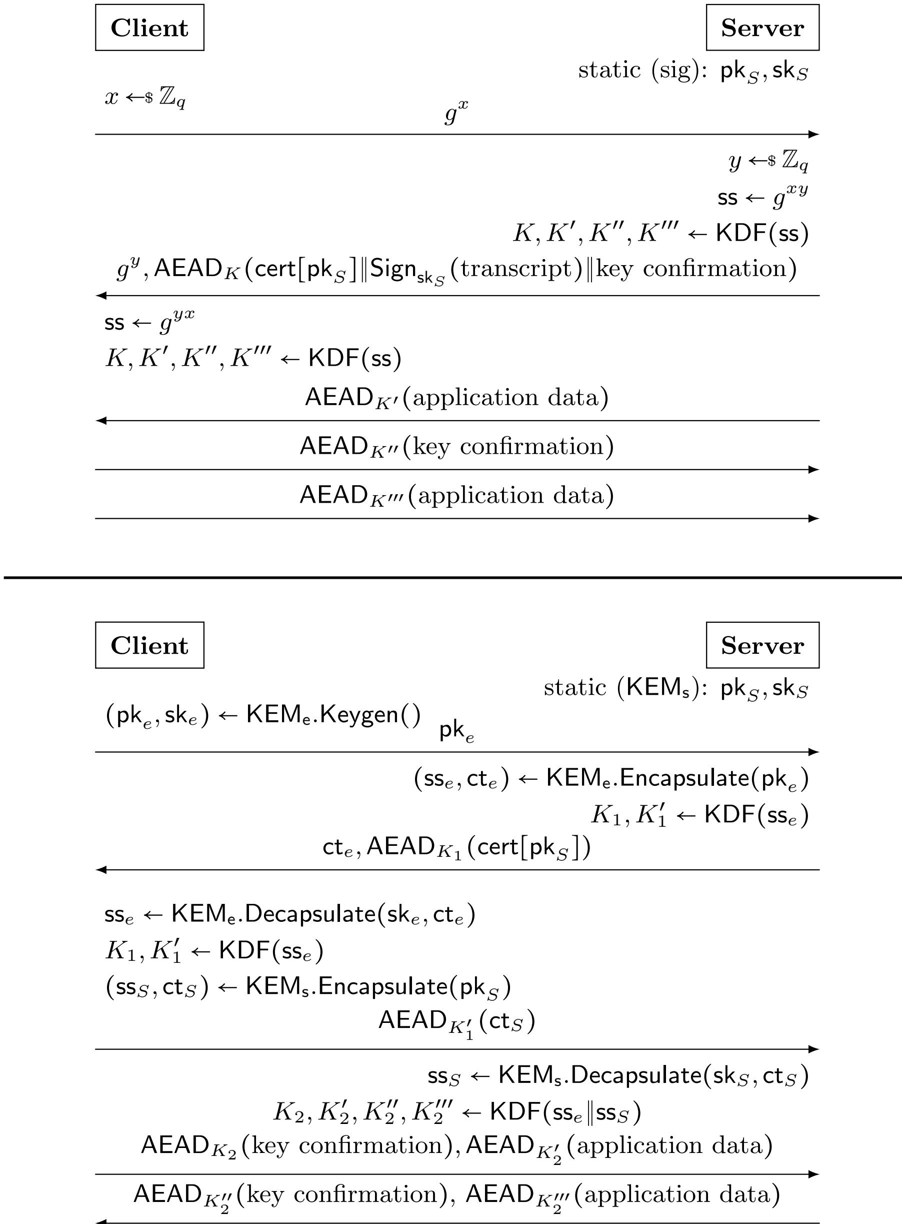


Figure 5.1: Simplified protocol flow diagrams of: (top) the TLS 1.3 handshake, using signatures for server authentication; and (bottom) the KEMTLS handshake, using KEMs for server authentication.

be done via Diffie–Hellman key exchange. However, as observed by Kuhnen [Kuh18], OPTLS requires the non-interactive key exchange properties of DH, which KEMs do not offer. To authenticate a server via KEM, the client encapsulates a ciphertext to the long-term public key contained in the server's certificate. This would naively result in a 2-RTT protocol. KEMTLS avoids the performance penalty this would imply by observing that, in many applications including HTTP, for a *useful* response from the server, it first needs to receive a request from the client. For example, which page the client is requesting from the server. KEMTLS allows the client to send its request in the same flow as it would in TLS 1.3, returning the protocol to 1.5-RTT. This is achieved by encrypting the data with a key that is derived from both the ephemeral key exchange and the shared secret encapsulated to the server's long-term key. This key is *implicitly* authenticated, as the client can not be sure of the server's presence before it receives a message (`ServerFinished`) in which the server uses the encapsulated secret. The right-hand side of Figure 5.1 describes a simplified version of a (unilaterally authenticated) KEMTLS handshake.

5.3 Experimental Setup

The following section describes the experimental setup used to acquire our results. Both protocols were benchmarked for handshake times, run-times of algorithms, peak memory usage, code size, and network traffic. Handshake times were measured in three network environments relevant to the IoT domain. This includes regular "broadband" internet, as well as two low-power wide-area network standards, LTE-Machine Type Communication (LTE-M) and Narrowband-IoT (NB-IoT), developed by the 3rd Generation network Partnership Project (3GPP). We give the characteristics employed for these environments in Table 5.1. While the performance characteristics of LTE-M and NB-IoT are based on numbers of the 3GPP [3rd15], the broadband scenario is based on realistic round-trip times of client-to-cloud communication within West Europe using a consumer-grade connection [PKLN21].

Table 5.1: Connection Characteristics according to 3GPP [3rd15]

Name	**Abbrev.**	**Bandwidth**	**RTT time**
Broadband	BB	1 Mbit	26 ms
LTE Machine Type	LTE-M	1 Mbit	120 ms
Narrowband-IoT	NB-IoT	46 kbit	3 s

Cryptographic Primitives

As KEMTLS is a post-quantum protocol, it is not specifically designed for transitional security. Although KEMTLS does not preclude their use, we do not

consider mixed classic/post-quantum certificates or hybrid (post-quantum plus elliptic curve) key-exchange methods in our experiments. For comparability, our PQTLS implementation is also exclusively using post-quantum algorithms. We evaluated all combinations of NIST finalists, except for the KEM *Classic McEliece*. Classic McEliece's public keys are too large to fit into memory and do not fit in the `ClientHello`'s `KeyShareEntry` extension [Res18, Sec. 4.2.8].

Both KEMTLS and PQTLS make use of a certificate authority (CA) that signs certificates. The CA's certificate, containing the CA's public key used for signature verification, is stored on the client device. Only leaf certificates, transmitted by the server during the handshake, differ in KEMTLS and TLS 1.3. For PQTLS they include the public key of a signature algorithm, in KEMTLS a KEM public key.

We only evaluate primitives at the lowest security level, NIST level I. These are the smallest and most efficient parameter sets.

5.3.1 Implementation

All benchmarks were conducted on a Silicon Labs STK3701A board, also known as the "Giant Gecko". This board was chosen because it features a 72 MHz ARM Cortex-M4F embedded processor and offers large enough memory (2 MB flash storage, 512 kB SRAM) to fit Rainbow public keys. As Cortex-M4 is the designated NIST PQC reference platform for embedded devices, there are optimized assembly implementations available for most finalist algorithms. The PQM4 project collects these implementations and provides extensive benchmarks [KRSS]. All PQC implementations used for benchmarking were taken from the PQM4 project. Only minor modifications, such as adding *verify* functions to signature schemes, fixing alignment issues and name-spacing symbol names had to be conducted. The code was compiled using GCC version 11.1, with the `-O3` speed optimization flag. In contrast to experiments run within the PQM4 project, we do not clock down the processor to avoid wait states. Instead, the processor runs at full speed. This makes sense since we are not exclusively interested in the run times of the primitives, but the performance of the overall system. Running the processor at full speed makes the PQC algorithms consume more cycles due to flash wait states and higher costs of memory accesses. However, since the PQC algorithms do not consume more wall-clock time, the actual handshake durations are not negatively affected. The Giant Gecko board exclusively takes on the role of an embedded (KEM)TLS client, wanting to connect to a backend server. To validate certificates sent in the handshake, we flash the CA's root certificate into the Giant Gecko's persistent memory during setup. For efficiency, the CA directly signs the server's certificate. This avoids the need for transmitting intermediate CAs, reducing the size of the certificate chain. As both endpoints in embedded scenarios are usually under some level of manufacturer control, this is a common deployment. Communication to the backend server is done via the Giant Gecko's Ethernet port, which is directly connected to a high-end computer. This host

computer simulates different network environments by using Linux's *netem* network emulation framework [HLP11]. The network emulation framework is set up to throttle bandwidth and delay round trip times (RTTs) according to the aforementioned network environments. Wiggers' original KEMTLS implementation [SSW20, SSW21] is used as server software. When running an iteration of the experiment, the corresponding PQC algorithms and the CA certificate are linked into the binary using Zephyr's *West* build tool. We then flash the binary onto the board via JLink. Benchmark results are received via serial communication.

Platform

To have a realistic setup, we employed a typical embedded systems software stack. In our case that includes an embedded real-time operating system (RTOS) with an open-source TCP/IP stack and added TLS support. For reproducibility, we used the Apache-licensed Zephyr RTOS [Zep]. Zephyr supports over 200 boards and is backed by the Linux Foundation and multiple large corporations involved with developing embedded systems, such as NXP, NORDIC or Memfault. It provides its own optimized embedded network stack and allows cycle-accurate run time measurements (given a board's hardware supports it). Our application code runs as the exclusive Zephyr thread, eliminating scheduling costs. PQTLS and KEMTLS support was added to the operating system via a custom WolfSSL module. All code, including reproducible build system and server software, used in this work is publicly available. KEMTLS certificate generation was conducted using a customized Python script based on Wiggers et al. code. Post-quantum certificates for PQTLS were generated using a fork of OpenSSL's command-line tool maintained by the Open Quantum Safe project [Opea]. The TLS 1.3 cipher suite `TLS_CHACHA20_POLY1305_SHA256` was used in all experiments.

WolfSSL Integration

Previous work [BSKNS20, GLF+21] also uses WolfSSL for running benchmarks on embedded systems. We decided to use WolfSSL for the same reasons as the mentioned works and to make comparisons with our results easier. WolfSSL is designed to be memory efficient and fast on embedded systems. On top, it already supports TLS 1.3 and has a clean implementation of TLS's state machine. This makes it an ideal basis for implementing PQTLS and KEMTLS. Adding post-quantum algorithms to WolfSSL is straightforward. WolfSSL's crypto provider, called WolfCrypt, has a clean API that can be extended easily. As the KEM Kyber was already included in WolfSSL by [BSKNS20], we did not need to make changes to the TLS 1.3 state machine. Apart from including the relevant ASN.1 object identifiers for KEMs and post-quantum signatures, only small changes such as increasing the maximum size of certificates had to be applied. Our embedded KEMTLS implementation is based on the same WolfSSL version as our PQTLS implementation. The majority of

the code is identical in the PQTLS and KEMTLS implementation. However, adding support for KEMTLS to WolfSSL still required significant effort. Apart from altering the certificate/ASN.1 parser to allow KEM keys in certificates (and using those), WolfSSL's internal state machine, key derivations and state structures had to be modified. In both our PQTLS and KEMTLS experiments the client only performs signature verification, so no code for signing was linked into the final binary.

5.4 Results

For developers of embedded systems, the trade-offs between ROM (code size), RAM (memory usage), network traffic and CPU time (run-time of code) are most crucial. In this section, we present our findings regarding the consumption of these resources by KEMTLS and TLS 1.3 using NIST PQC finalists.

The run-time of algorithms impacts the device's energy consumption. This is especially relevant for battery-powered devices that rely on the possibility to hibernate when inactive. Network traffic also affects energy consumption, as operating an antenna is usually a very energy-consuming operation. Depending on the underlying wireless technology, network traffic can also be expensive in terms of network provider fees. Our results are representative for Cortex-M4-based platforms in general. Hence we focus on benchmarks that are independent of our specific evaluation board. As energy consumption varies heavily based on a board's design, choice of peripherals and transmission technology we did not include direct energy measurements into our results. Instead, we present code size, consumed memory, handshake traffic, handshake duration and run-time of PQC primitives. All KEMTLS and PQTLS instantiations were run 1000 times, with each run using a different CA and leaf certificate. The presented benchmarks are averaged over all runs. The NIST signature finalist Rainbow, which is included as a representative for multivariate-based cryptography, is only present in the KEMTLS results. This is because Rainbow public keys are very large. There was not enough memory to fit Rainbow as well as another signature scheme. It could therefore not be included into the PQTLS benchmarks. We emphasize that all employed PQC algorithms were optimized for speed, and not stack consumption.

5.4.1 Storage and Memory Consumption

Both protocol implementations are roughly the same size. Without post-quantum primitives, they have a code size of around 111 kB. Table 5.2 shows combinations of PQC algorithms with their measured code size. For KEMTLS, only instantiations with one KEM used for both ephemeral key exchange and authentication are shown. Including two KEMs does not give an advantage, but increases code size. However, for completeness, an overview with all combinations can be found in Table 5.4. Similarly, PQTLS instantiations with the

same signature algorithm used for CA and leaf certificates are shown. Additionally, we include the combination of Dilithium and Falcon, where Dilithium is used as the handshake signature algorithm. This combination was suggested by Sikideris et al. [SKD20] to make use of Dilithiums faster signing times for servers without hardware support for Falcon's double-precision floating-point operations.

The table also shows the PQC code's share of the overall code size as a percentage. Also included in Table 5.2 is memory consumption. Shown is the peak of consumed memory, in both heap and stack, during the handshake. This includes the memory consumed by the protocol implementation and PQC primitives.

In contrast to TLS 1.3, KEMTLS uses a KEM encapsulation instead of a signature verification to authenticate the connection. KEMTLS, therefore, needs code for KEM encapsulation, whereas TLS 1.3 does not. TLS 1.3 on the other hand needs the code for two distinct verification algorithms if different signature algorithms are used for CA and leaf certificates. Instantiations with NTRU ephemeral key exchange are notable outliers in terms of code size, requiring over 200 kB of code. This is in line with results reported by PQM4 [KRSS]. Interestingly, this big increase in code size can not be observed when NTRU is used exclusively for authentication. This is because the client requires key generation and decapsulation code for ephemeral key exchange, whereas authentication via KEM only requires encapsulation functionality. Whenever Rainbow is used, the CA certificate containing a Rainbow public key takes up between 33% and 53% of the overall consumed storage space. This, however, does not disqualify Rainbow from usage on embedded systems, due to its small signature and very fast verification times (see Section 5.4.2).

Further, the results show that the lattice-based schemes perform well in terms of memory consumption. The consumed memory is mainly driven by stack usage of the PQC signature algorithms. Only Rainbow is an exception here. With a Rainbow-powered CA certificate, the very large public key has to be loaded into memory and held during signature verification. This requires a large allocation of heap space. In a custom certificate loader implementation it would be possible to store the public key in an already usable form in flash. Then the public key could directly be streamed in from flash (similar to [GHK+21]), without the need to hold it in memory entirely. However, since we present comparable results of reusable code, we did not include this kind of optimization for an individual algorithm.

5.4.2 Handshake Times

Apart from storage and memory consumption, handshake times are key in an embedded environment. Table 5.3 shows handshake times for different transmission technologies measured in millions of cycles. All possible instantiations are presented in Table 5.5. In Figure 5.2 we show the handshake times and traffic for the broadband and NB-IoT scenarios. In a real deployment, the de-

Table 5.2: Code and CA certificate sizes (and as percentage of total ROM size), and peak memory usage in the experiments. Parameter sets used are NIST level I.

	KEX	**Auth.**	**CA**	**PQC code**	**(%)**	**CA size**	**(%)**	**Memory**
KEMTLS	Kyber	Kyber	Dilithium	29.0 kB	(20.1%)	3.9 kB	(2.7%)	49.7 kB
	Kyber	Kyber	Falcon	25.7 kB	(18.6%)	1.7 kB	(1.2%)	52.8 kB
	Kyber	Kyber	Rainbow	29.8 kB	(9.8%)	161.8 kB	(53.4%)	167.0 kB
	NTRU	NTRU	Dilithium	203.4 kB	(63.9%)	3.9 kB	(1.2%)	49.7 kB
	NTRU	NTRU	Falcon	200.0 kB	(63.9%)	1.7 kB	(0.6%)	52.8 kB
	NTRU	NTRU	Rainbow	204.0 kB	(42.8%)	161.8 kB	(33.9%)	182.9 kB
	SABER	SABER	Dilithium	31.5 kB	(21.5%)	3.9 kB	(2.7%)	49.7 kB
	SABER	SABER	Falcon	28.2 kB	(20.0%)	1.7 kB	(1.2%)	52.8 kB
	SABER	SABER	Rainbow	32.2 kB	(10.5%)	161.8 kB	(53.0%)	167.9 kB
PQTLS	Kyber	Dilithium	Dilithium	29.0 kB	(20.1%)	4.0 kB	(2.8%)	58.0 kB
	Kyber	Falcon	Dilithium	34.4 kB	(23.0%)	4.0 kB	(2.7%)	60.7 kB
	Kyber	Falcon	Falcon	25.8 kB	(18.6%)	1.8 kB	(1.3%)	56.2 kB
	NTRU	Dilithium	Dilithium	203.4 kB	(63.8%)	4.0 kB	(1.3%)	56.6 kB
	NTRU	Falcon	Dilithium	208.7 kB	(64.4%)	4.0 kB	(1.2%)	59.3 kB
	NTRU	Falcon	Falcon	200.1 kB	(63.9%)	1.8 kB	(0.6%)	54.8 kB
	SABER	Dilithium	Dilithium	31.5 kB	(21.5%)	4.0 kB	(2.7%)	58.0 kB
	SABER	Falcon	Dilithium	36.8 kB	(24.2%)	4.0 kB	(2.6%)	60.7 kB
	SABER	Falcon	Falcon	28.2 kB	(20.0%)	1.8 kB	(1.3%)	56.2 kB

Remark: NTRU, Rainbow and SABER have been eliminated from the NIST selection.

vice would likely go into a low power mode or sleep instead of actively polling data during a slow transmission. This behavior however depends highly on the specifics of the embedded system and its transmission technology. Therefore, to achieve reproducible results, the CPU was running at a constant speed of 72MHz during all experiments. This also makes a direct translation to wall time possible. The table also shows the percentage of cycles spent on the underlying PQC primitives. The remaining cycles are spent in the TLS state machine, memory operations or waiting for I/O.

Time spent in crypto operations is significant in the broadband and LTE-M setting. Whereas the NB-IoT transmission is so slow, that the share of cycles spent in cryptographic operations is very low (0.8% - 1.7%). In low-bandwidth/high-RTT settings like NB-IoT, the transmission size of certificates and public keys is the main driving factor of run time. Loading large public keys from storage into memory is a relevant factor as well, slowing down the otherwise fast Rainbow signature algorithm. Cycles spent to access memory and storage also become increasingly negligible when using slow transportation mediums. This is visible in Figure 5.2b, where the instantiations with similarly sized handshake traffic clearly form clusters.

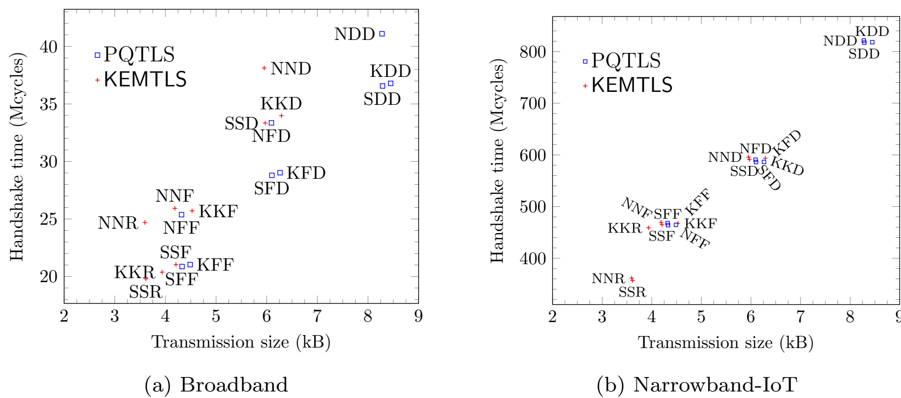
Table 5.3: TLS handshake traffic and runtime for various scenarios. Parameter sets used are NIST level I.

				Handshake	**Handshake time in Mcycles (% of crypto)**		
	KEX	**Auth.**	**CA**	**traffic**	**BB (%)**	**LTE-M (%)**	**NB-IoT (%)**
KEMTLS	Kyber	Kyber	Dilithium	6.3 kB	17.1 (30.2%)	34.0 (15.2%)	593.6 (0.9%)
	Kyber	Kyber	Falcon	4.5 kB	12.3 (27.2%)	25.7 (13.0%)	467.8 (0.7%)
	Kyber	Kyber	Rainbow	3.9 kB	11.3 (25.1%)	20.4 (13.9%)	459.0 (0.6%)
	NTRU	NTRU	Dilithium	6.0 kB	21.3 (46.0%)	38.1 (25.6%)	595.8 (1.6%)
	NTRU	NTRU	Falcon	4.2 kB	16.6 (47.8%)	25.9 (30.6%)	469.7 (1.7%)
	NTRU	NTRU	Rainbow	3.6 kB	15.7 (47.4%)	24.7 (30.1%)	361.6 (2.1%)
	SABER	SABER	Dilithium	6.0 kB	16.3 (29.4%)	33.3 (14.4%)	590.8 (0.8%)
	SABER	SABER	Falcon	4.2 kB	11.6 (25.5%)	21.0 (14.1%)	464.8 (0.6%)
	SABER	SABER	Rainbow	3.6 kB	10.7 (23.1%)	19.8 (12.5%)	356.8 (0.7%)
PQTLS	Kyber	Dilithium	Dilithium	8.4 kB	19.9 (35.9%)	36.8 (19.5%)	818.1 (0.9%)
	Kyber	Falcon	Dilithium	6.3 kB	15.5 (33.0%)	29.0 (17.6%)	586.4 (0.9%)
	Kyber	Falcon	Falcon	4.5 kB	10.9 (30.1%)	21.0 (15.6%)	464.6 (0.7%)
	NTRU	Dilithium	Dilithium	8.3 kB	24.3 (47.6%)	41.1 (28.1%)	821.3 (1.4%)
	NTRU	Falcon	Dilithium	6.1 kB	19.9 (47.8%)	33.4 (28.5%)	590.6 (1.6%)
	NTRU	Falcon	Falcon	4.3 kB	15.2 (50.3%)	25.4 (30.2%)	468.0 (1.6%)
	SABER	Dilithium	Dilithium	8.3 kB	19.7 (35.2%)	36.6 (19.0%)	817.3 (0.8%)
	SABER	Falcon	Dilithium	6.1 kB	15.3 (32.0%)	28.8 (17.0%)	586.2 (0.8%)
	SABER	Falcon	Falcon	4.3 kB	10.7 (28.5%)	20.9 (14.6%)	464.0 (0.7%)

Remark: NTRU, Rainbow and SABER have been eliminated from the NIST selection.

Both PQTLS and KEMTLS use a KEM for key exchange. While the performance of the module lattice KEMs Kyber and SABER is similar, they both outperform NTRU for this task. This is mainly due to the rather slow key generation of NTRU increasing handshake time. Slow key generation is also the reason why PQTLS and KEMTLS instantiations using NTRU have the highest

percentage of cycles spent in PQC operations.



(a) Broadband (b) Narrowband-IoT

Figure 5.2: Handshake times and traffic for instantiations of KEMTLS and PQTLS. Letters represent the algorithms Dilithium, Falcon, Kyber, NTRU, Rainbow, and SABER in the roles of ephemeral key exchange, handshake authentication and CA, in that order.

All KEMs outperform Dilithium when used for authentication. This makes sense as Dilithium's verify routine is slower than the encapsulation routine of all investigated KEMs. Dilithium's performance also suffers from its large public key and signature that increase the required transmission size. In slow, bandwidth-constrained network environments, such as NB-IoT, this drawback becomes even more apparent. Rainbow performs well in terms of handshake times when used as a CA certificate. Not only because it has a fast, bitsliced Cortex-M4 implementation. Since the large Rainbow public key is stored on the client device, only the small signature has to be transmitted during the handshake. Rainbow's small signature and fast runtime make it a good fit for CA certificates if the storage and memory demands can be afforded. The instantiations with Rainbow offer the fastest KEMTLS handshake times throughout all transmission mediums. Additionally, the shortest NB-IoT handshake times use KEMTLS with Rainbow and SABER. Falcon on the other hand performs very well on the Cortex-M4 platform in our experiments. In terms of runtime, it even outperforms KEMs for server authentication. However, this is only true for the client side. Signing operations using Falcon are considerably more expensive than KEM decapsulations. But these operations are conducted on the server side, increasing server load, which is not part of our measurements. Additionally, Falcon's public key and signature sizes are comparable to the sizes of the KEM's public keys and ciphertexts. So it is not surprising that PQTLS instantiations using Falcon perform well. In the broadband and LTE-M setting, PQTLS with Falcon and SABER performs as well as KEMTLS with Rainbow and SABER.

Table 5.4: Code and CA certificate sizes (and as percentage of total ROM size), and peak memory usage in the experiments.

	KEX	Auth.	CA	PQC code	(%)	CA size	(%)	Memory
KEMTLS	Kyber	Kyber	Dilithium	29.0 kB	(20.1%)	3.9 kB	(2.7%)	49.7 kB
	Kyber	Kyber	Falcon	25.7 kB	(18.6%)	1.7 kB	(1.2%)	52.8 kB
	Kyber	Kyber	Rainbow	29.8 kB	(9.8%)	161.8 kB	(53.4%)	167.0 kB
	Kyber	NTRU	Dilithium	41.0 kB	(26.3%)	3.9 kB	(2.5%)	49.7 kB
	Kyber	NTRU	Falcon	37.7 kB	(25.0%)	1.7 kB	(1.1%)	52.8 kB
	Kyber	NTRU	Rainbow	41.7 kB	(13.3%)	161.8 kB	(51.4%)	182.9 kB
	Kyber	SABER	Dilithium	44.9 kB	(28.1%)	3.9 kB	(2.4%)	49.7 kB
	Kyber	SABER	Falcon	41.7 kB	(26.9%)	1.7 kB	(1.1%)	52.8 kB
	Kyber	SABER	Rainbow	45.7 kB	(14.3%)	161.8 kB	(50.8%)	167.9 kB
	NTRU	Kyber	Dilithium	216.3 kB	(65.3%)	3.9 kB	(1.2%)	49.7 kB
	NTRU	Kyber	Falcon	213.0 kB	(65.4%)	1.7 kB	(0.5%)	52.8 kB
	NTRU	Kyber	Rainbow	217.1 kB	(44.3%)	161.8 kB	(33.0%)	182.9 kB
	NTRU	NTRU	Dilithium	203.4 kB	(63.9%)	3.9 kB	(1.2%)	49.7 kB
	NTRU	NTRU	Falcon	200.0 kB	(63.9%)	1.7 kB	(0.6%)	52.8 kB
	NTRU	NTRU	Rainbow	204.0 kB	(42.8%)	161.8 kB	(33.9%)	182.9 kB
	NTRU	SABER	Dilithium	219.7 kB	(65.6%)	3.9 kB	(1.2%)	49.7 kB
	NTRU	SABER	Falcon	216.4 kB	(65.7%)	1.7 kB	(0.5%)	52.8 kB
	NTRU	SABER	Rainbow	220.4 kB	(44.7%)	161.8 kB	(32.8%)	182.9 kB
	SABER	Kyber	Dilithium	44.5 kB	(27.9%)	3.9 kB	(2.4%)	49.7 kB
	SABER	Kyber	Falcon	41.3 kB	(26.8%)	1.7 kB	(1.1%)	52.8 kB
	SABER	Kyber	Rainbow	45.3 kB	(14.2%)	161.8 kB	(50.8%)	167.9 kB
	SABER	NTRU	Dilithium	43.9 kB	(27.6%)	3.9 kB	(2.5%)	49.7 kB
	SABER	NTRU	Falcon	40.6 kB	(26.4%)	1.7 kB	(1.1%)	52.8 kB
	SABER	NTRU	Rainbow	44.6 kB	(14.0%)	161.8 kB	(50.9%)	182.9 kB
	SABER	SABER	Dilithium	31.5 kB	(21.5%)	3.9 kB	(2.7%)	49.7 kB
	SABER	SABER	Falcon	28.2 kB	(20.0%)	1.7 kB	(1.2%)	52.8 kB
	SABER	SABER	Rainbow	32.2 kB	(10.5%)	161.8 kB	(53.0%)	167.9 kB
PQTLS	Kyber	Dilithium	Dilithium	29.0 kB	(20.1%)	4.0 kB	(2.8%)	58.0 kB
	Kyber	Dilithium	Falcon	34.4 kB	(23.3%)	1.8 kB	(1.2%)	60.0 kB
	Kyber	Falcon	Dilithium	34.4 kB	(23.0%)	4.0 kB	(2.7%)	60.7 kB
	Kyber	Falcon	Falcon	25.8 kB	(18.6%)	1.8 kB	(1.3%)	56.2 kB
	NTRU	Dilithium	Dilithium	203.4 kB	(63.8%)	4.0 kB	(1.3%)	56.6 kB
	NTRU	Dilithium	Falcon	208.7 kB	(64.9%)	1.8 kB	(0.6%)	58.6 kB
	NTRU	Falcon	Dilithium	208.7 kB	(64.4%)	4.0 kB	(1.2%)	59.3 kB
	NTRU	Falcon	Falcon	200.1 kB	(63.9%)	1.8 kB	(0.6%)	54.8 kB
	SABER	Dilithium	Dilithium	31.5 kB	(21.5%)	4.0 kB	(2.7%)	58.0 kB
	SABER	Dilithium	Falcon	36.8 kB	(24.6%)	1.8 kB	(1.2%)	60.0 kB
	SABER	Falcon	Dilithium	36.8 kB	(24.2%)	4.0 kB	(2.6%)	60.7 kB
	SABER	Falcon	Falcon	28.2 kB	(20.0%)	1.8 kB	(1.3%)	56.2 kB

Remark: NTRU, Rainbow and SABER have been eliminated from the NIST selection.

Table 5.5: TLS handshake traffic and runtime for various scenarios

	KEX	Auth.	CA	Handshake traffic	Handshake time in Mcycles (% of crypto)		
					BB (%)	**LTE-M (%)**	**NB-IoT (%)**
KEMTLS	Kyber	Kyber	Dilithium	6.3 kB	17.1 (30.2%)	34.0 (15.2%)	593.6 (0.9%)
	Kyber	Kyber	Falcon	4.5 kB	12.3 (27.2%)	25.7 (13.0%)	467.8 (0.7%)
	Kyber	Kyber	Rainbow	3.9 kB	11.3 (25.1%)	20.4 (13.9%)	459.0 (0.6%)
	Kyber	NTRU	Dilithium	6.1 kB	17.1 (31.5%)	34.1 (15.8%)	592.2 (0.9%)
	Kyber	NTRU	Falcon	4.4 kB	12.4 (28.8%)	21.7 (16.4%)	466.2 (0.8%)
	Kyber	NTRU	Rainbow	3.8 kB	11.4 (27.0%)	20.5 (15.0%)	358.1 (0.9%)
	Kyber	SABER	Dilithium	6.1 kB	16.8 (30.0%)	33.6 (15.0%)	591.5 (0.8%)
	Kyber	SABER	Falcon	4.4 kB	12.0 (26.6%)	21.5 (14.9%)	465.4 (0.7%)
	Kyber	SABER	Rainbow	3.8 kB	11.0 (24.5%)	20.2 (13.4%)	357.4 (0.8%)
	NTRU	Kyber	Dilithium	6.1 kB	21.3 (44.8%)	38.2 (25.0%)	596.9 (1.6%)
	NTRU	Kyber	Falcon	4.4 kB	16.6 (46.4%)	25.9 (29.7%)	470.8 (1.6%)
	NTRU	Kyber	Rainbow	3.8 kB	15.5 (46.3%)	24.7 (29.1%)	462.4 (1.6%)
	NTRU	NTRU	Dilithium	6.0 kB	21.3 (46.0%)	38.1 (25.6%)	595.8 (1.6%)
	NTRU	NTRU	Falcon	4.2 kB	16.6 (47.8%)	25.9 (30.6%)	469.7 (1.7%)
	NTRU	NTRU	Rainbow	3.6 kB	15.7 (47.4%)	24.7 (30.1%)	361.6 (2.1%)
	NTRU	SABER	Dilithium	6.0 kB	20.8 (45.1%)	37.7 (24.9%)	594.9 (1.6%)
	NTRU	SABER	Falcon	4.2 kB	16.2 (46.6%)	25.6 (29.5%)	468.9 (1.6%)
	NTRU	SABER	Rainbow	3.6 kB	15.3 (46.3%)	24.3 (29.1%)	360.9 (2.0%)
	SABER	Kyber	Dilithium	6.1 kB	16.8 (29.4%)	33.6 (14.7%)	593.0 (0.8%)
	SABER	Kyber	Falcon	4.4 kB	11.9 (26.1%)	22.7 (13.7%)	466.8 (0.7%)
	SABER	Kyber	Rainbow	3.8 kB	11.0 (23.7%)	20.2 (12.8%)	458.3 (0.6%)
	SABER	NTRU	Dilithium	6.0 kB	16.8 (30.8%)	33.7 (15.3%)	591.5 (0.9%)
	SABER	NTRU	Falcon	4.2 kB	12.0 (27.9%)	21.5 (15.5%)	465.6 (0.7%)
	SABER	NTRU	Rainbow	3.6 kB	11.0 (25.8%)	20.2 (14.1%)	357.6 (0.8%)
	SABER	SABER	Dilithium	6.0 kB	16.3 (29.4%)	33.3 (14.4%)	590.8 (0.8%)
	SABER	SABER	Falcon	4.2 kB	11.6 (25.5%)	21.0 (14.1%)	464.8 (0.6%)
	SABER	SABER	Rainbow	3.6 kB	10.7 (23.1%)	19.8 (12.5%)	356.8 (0.7%)
PQTLS	Kyber	Dilithium	Dilithium	8.4 kB	19.9 (35.9%)	36.8 (19.5%)	818.1 (0.9%)
	Kyber	Dilithium	Falcon	6.7 kB	14.7 (35.4%)	31.0 (16.8%)	595.8 (0.9%)
	Kyber	Falcon	Dilithium	6.3 kB	15.5 (33.0%)	29.0 (17.6%)	586.4 (0.9%)
	Kyber	Falcon	Falcon	4.5 kB	10.9 (30.1%)	21.0 (15.6%)	464.6 (0.7%)
	NTRU	Dilithium	Dilithium	8.3 kB	24.3 (47.6%)	41.1 (28.1%)	821.3 (1.4%)
	NTRU	Dilithium	Falcon	6.5 kB	19.0 (50.3%)	35.3 (27.2%)	599.2 (1.6%)
	NTRU	Falcon	Dilithium	6.1 kB	19.9 (47.8%)	33.4 (28.5%)	590.6 (1.6%)
	NTRU	Falcon	Falcon	4.3 kB	15.2 (50.3%)	25.4 (30.2%)	468.0 (1.6%)
	SABER	Dilithium	Dilithium	8.3 kB	19.7 (35.2%)	36.6 (19.0%)	817.3 (0.8%)
	SABER	Dilithium	Falcon	6.5 kB	14.5 (34.2%)	30.7 (16.2%)	595.2 (0.8%)
	SABER	Falcon	Dilithium	6.1 kB	15.3 (32.0%)	28.8 (17.0%)	586.2 (0.8%)
	SABER	Falcon	Falcon	4.3 kB	10.7 (28.5%)	20.9 (14.6%)	464.0 (0.7%)

Remark: NTRU, Rainbow and SABER have been eliminated from the NIST selection.

5.5 Discussion

Our results show that KEMTLS with server-only authentication uses less memory than PQTLS and has similar code sizes. Due to Falcon's verification algorithm being very efficient, in terms of bandwidth and computation time, PQTLS with Falcon performs as well as or better than any KEMTLS instantiation. The only exception are the KEMTLS instantiations using SABER or NTRU with Rainbow, where the ability of KEMTLS to use Rainbow due to lower memory usage saves a few bytes and thus become the best-performing instantiations in the NB-IoT scenario. Falcon also performs better than Dilithium on the client side, in any scenario.

Although we have not measured client authentication or an embedded server, we can extrapolate from our results. As reported by PQM4 [KRSS] and Sikideris et al. [SKD20], Falcon's signing algorithm, especially without hardware support, is significantly more costly than Dilithium's or any of the KEM operations. This suggests that Falcon is perhaps not generically suitable for post-quantum authentication.

Sikideris et al. also suggested a combination of Dilithium and Falcon for PQTLS, in scenarios where there is no hardware support for Falcon's double-precision floating-point operations. Dilithium would be put in the leaf certificate, to make use of its efficient signing times for online handshake signatures. Falcon's smaller public key and signature sizes would be beneficial for the CA certificate algorithm, which signs the leaf certificate only once, but the signature is transmitted many times. However, our results show that for embedded clients that only need to do signature validation Falcon is preferable over Dilithium, especially in very low bandwidth scenarios like NB-IoT.

5.6 Conclusion and Future Work

In this chapter, we compared the performance of KEMTLS and TLS 1.3 using NIST PQC finalists in an embedded environment. This environment was represented by a Cortex-M4-based client communicating with a desktop-class server. We showed that a KEMTLS client consumes less memory than TLS 1.3, due to the smaller memory footprint of KEMs. The code size did not differ between KEMTLS and TLS 1.3. Since only server authentication was used, both protocols require a signature verify function and KEM for key exchange. Our run times show that in both protocols PQC primitives require a significant amount of computational time during the handshake, sometimes requiring over 50% of the entire handshake time. Even in the LTE-M setting, the percentage of cycles spent in PQC computations is considerable. However, in the bandwidth-constrained NB-IoT setting, handshake times are mostly driven by handshake size. In these conditions, Rainbow's very small signatures are an advantage. While Dilithium is generally outperformed by KEMs when used for authentication, Falcon performs very well due to its efficient verification algorithm.

However, signing in Falcon is a very costly operation. Future work should therefore investigate KEMTLS and TLS 1.3 using client authentication, and embedded KEMTLS and post-quantum TLS 1.3 servers. In both of these applications, the embedded TLS 1.3 client needs to produce handshake signatures. This would increase the cost of using signatures instead of KEMs significantly, leading to new trade-offs. Another avenue of research is the pre-distributed key setting, where the client already knows the server's public key. In this setting, bandwidth can be reduced even further, which may be compelling for the NB-IoT application.

Chapter 6

Stateless Hash-Based Signatures for Post-Quantum Security Keys

This chapter is based on work originally presented in the paper:

> Ruben Gonzalez. Stateless hash-based signatures for post-quantum security keys. In *International Conference on Applied Cryptography and Network Security*, volume 15654 of *LNCS*. Springer, 2025. `https://eprint.iacr.org/2025/298`

In it, we[1] continue to investigate post-quantum authentication in resource-constrained environments. In contrast to the previous chapter, this chapter is concerned with user authentication. FIDO2 can be used to facilitate secure user authentication within an application that provides server authentication (as seen in the previous chapter). It leverages asymmetric signatures for that purpose. Two of the three NIST-selected signature algorithms are lattice based, offering good performance but posing challenges due to complex implementation and intricate security assumptions. A more conservative choice for quantum-safe authentication are hash-based signature systems, such as the also-standardized SLH-DSA, which is based on the SPHINCS$^+$ construction. However, due to large signature sizes and low signing speeds, hash-based systems have only found use in niche applications. In this chapter we combine different approaches to show that the SPHINCS$^+$ signature system can be optimized in its parameters and implementation to be high performing, even when signing in an embedded setting. We demonstrate this in the context of user

[1] For consistency with the other chapters, "we" is used also for single-author efforts.

authentication using hardware security keys within FIDO. Due to conservative security assumptions, our solution does not require a hybrid construction and can be used as a drop-in replacement to perform authentication on current security keys. For reproducibility and to encourage further research, we publish our implementation and a framework for benchmarking tailored SPHINCS⁺ instantiations on Cortex-M4 MCUs.

As this is a very recent publication, no parts are outdated at the time of writing.

Contribution. We show that SPHINCS⁺ signature systems can be instantiated to perform well in embedded settings that require signing and relatively short signatures, even outperforming lattice-based solutions from the literature while remaining highly portable. To do so, we compare the performance to results previously acquired in the same setting (FIDO authentication) and on the same hardware (the nRF52840 development board). Furthermore, we show that in contrast to previous work utilizing lattice-based schemes, no complex implementation-level optimization tricks or hybrid constructions are necessary for our solution.

Associated Software. The software developed for this chapter includes a benchmarking framework capable of finding, instantiating, building and benchmarking SPHINCS⁺ libraries without dependency to the standard library. These SPHINCS⁺ libraries are based on the C reference implementation and can be used for experiments in embedded systems. It further includes a Rust wrapper for the SPHINCS⁺ libraries, acting as a compatibility layer to the TockOS-based firmware. A Client to Authenticator Protocol (CTAP) implementation is released for use in FIDO, based on Google's OpenSK, employing various adjusted SPHINCS⁺ instantiations. For portability and compatibility with OpenSK, the code is written in Rust. As part of the experiments conducted, the CryptoCell 310 hardware-based hash accelerator was used for comparison, but is not required to achieve the documented results. The released code can therefore be used across different platforms and embedded processor architectures. Additionally, we release a tool for quickly identifying suitable SPHINCS⁺ parameter sets for specific requirements, such as speed or signature size. All code needed to replicate the results is available at `http://doi.org/10.5281/zenodo.17381244`.

Organization of this chapter. After an introduction in Section 6.1 provides context on the significance of lattice-based and hash-based post-quantum algorithms in the embedded world, related works are presented. Section 6.2 then gives necessary background on FIDO-based authentication and post-quantum signatures, with a specific emphasis on hash-based signatures. The development of a SPHINCS⁺-based FIDO authenticator, detailing hardware and software requirements and parameter choices for custom SPHINCS⁺ instantiations, is documented in Section 6.3. Results are then described in Section 6.4, providing performance benchmarks of adjusted SPHINCS⁺ instantiations within FIDO and comparisons to existing, lattice-based solutions. Finally, the re-

sults and their implications are discussed in Section 6.4.5 and concluded in Section 6.5.

6.1 Introduction

Two out of three NIST-selected signature schemes rely on lattice-based constructions. That is not surprising, as lattice-based signatures offer high performance at comparatively small key and signature size. However, trust in these lattice-based PQC algorithms grows only slowly due to their novelty, complexity and tangled security assumptions [BBB+24, Fir]. Consequently, lattice-based algorithms are usually deployed in so-called *hybrid* mode, combining them with pre-quantum algorithms to ensure continued security against both classical and quantum adversaries. A class of PQC algorithms that do not suffer from this lack of trust are hash-based constructions [Lam79]. In fact, the only NIST standardized PQC algorithm that isn't based on structured-lattice constructions is the hash-based SLH-DSA signature system [Nat24c]. SLH-DSA is an instantiation of the SPHINCS+ signature framework [ABB+22]. A major drawback of SPHINCS+ is its slow signing speed and very large signature size. This seems to make it a rather poor choice for many applications, especially for applications that require signing in resource-constraint environments. On the other hand, hash-based systems offer the possibility of using PQC without hybrid constructions, reducing complexity and resource overhead.

In this work we adjust SPHINCS+, by carefully tuning its parameters and primitives, for a use case it might seem unfit for: FIDO. FIDO (Fast Identity Online) is an authentication standard for endusers, designed to avoid reliance on passwords and defeat phishing [FIDa]. We demonstrate that a SPHINCS+ signature system can be instantiated for use in a resource constrained FIDO security key, satisfying tough resource and time constrains. We further compare the performance and resource consumption of our implementation to previous FIDO experiments employing lattice-based hybrid PQC constructions and show that our SPHINCS+-based solution can outperform them.

6.1.1 Related Work

This section gives an overview of previous works on Post-Quantum FIDO and optimized SPHINCS+ instantiations.

Post-Quantum FIDO Implementation.

In [GKP+23] Ghinea et al. implement CTAP using the, now standardized [Nat24b], lattice-based CRYSTALS-Dilithium signature system for FIDO-based authentication. Their work implements a hybrid approach using the post-quantum Dilithium and pre-quantum ECDSA algorithms. The authors chose Dilithium over the other now standardized PQC signature

systems Falcon and SPHINCS$^+$ for three reasons: speed, complexity and size. Dilithium has a much faster key generation than Falcon. This is relevant in CTAP, as the keys have to be generated on the embedded device. Additionally, Dilithium has a less complex implementation than Falcon as it does not rely on floating-point arithmetic. SPHINCS$^+$ was not chosen for their experiment due to large signature sizes and poor signing performance. To reduce required storage space their implementation does not actually save a private key, but a small 32 byte seed that can be used to compute the private key on the fly. Moreover, they invest significant engineering effort to tweak their Dilithium implementation to recompute certain parts of the private key and intermediate results during every signing operation. This is done to reduce Dilithium's memory footprint, trading worse runtime for reduced memory consumption similar to [BRS22]. In their paper they also introduce requirements for runtime, memory consumption and message size within a FIDO security key setting. In our work we adhere to these requirements, making a direct comparison possible. As both implementations are based on OpenSK, we could reuse parts of their setup in our experiments.

SHPINCS+ Instantiations and Optimization.

SPHINCS$^+$ was submitted to the NIST PQC competition in 2017 [ABB$^+$22]. Since 2024 it is standardized in the Federal Information Processing Standard (FIPS) 205 under the name SLH-DSA. As over 90% of computation in SPHINCS$^+$ is spent inside the underlying hash function [KRSS], most research has focused on optimizing that hash function e.g. in hardware [Saa24a] [KJPL24]. Karl et al. recently analysed the impact of hardware-accelerated SPHINCS$^+$ and reviewing possible architectures for such hardware [KSS24]. The authors also include estimates of communication costs for using such hardware acceleration. In this work we complement that study, as we use hardware acceleration for a hash primitive. In another paper, Kölbl and Philipoom take note of the possibility to tweak SPHINCS$^+$ parameter sets for custom use cases [KP22]. They show that SPHINCS$^+$ signature sizes and verification speeds can be drastically reduced if the maximum amount of allowed signings per private key is reduced. While NIST mandated that 2^{64} signatures should be possible for a single private key without compromising security [Nat16], this threshold is unnecessarily high for many use cases. Based on this, Kölbl and Philipoom present a SPHINCS$^+$ parameterization that allows for fast verification on embedded devices and offers relatively short signatures. They show experimentally, that this enables firmware verification using SPHINCS$^+$ on an embedded device. Their parameters maintain compatibility to SLH-DSA, except for the requirement of allowed signatures per private key. The allowed signatures per key are drastically reduced to 2^{10} or 2^{20}, which is more than enough for firmware signing. Our work complements their work in the sense that it also approaches an embedded use case for which SPHINCS$^+$ seems unfit at first sight. However, their work

focuses exclusively on verification on an embedded device, which is arguable the much easier problem for SPHINCS$^+$. They further state: *"Compared to other post-quantum signature schemes like Dilithium, the signing speed will always be significantly worse"*. Our work tackles this more difficult case of key generation and signing within the embedded device.

6.2 Background

This section details the background needed to understand the implementation and results. First we describe FIDO, a widely-used, signature-based user authentication standard. We then detail the necessary background on PQC and hash-based signatures in particular.

6.2.1 FIDO-Based Authentication

The FIDO2 standard defines a signature-based and phishing-resistant user authentication mechanism. It can be used for single or second-factor authentication. As FIDO2 is a vast standard, this section details only its aspects relevant for this work. Within FIDO, users authenticate themselves by signing login data. For that purpose the user stores a public key in the application upon registration. During registration, the user either stores only the public key in the application, this is referred to as *resident key*[2] setting, or the symmetrically encrypted private key alongside it, which is referred to as *non-resident key* setting. Specifically, the FIDO standard explicitly allows to include the encrypted private key into metadata stored in the application. Figure 6.1 shows the high-level difference between the two options during authentication.

An advantage of the *non-resident key* setting is that the private key (amongst other metadata) does not need to be stored on the user's side. This is very relevant as FIDO otherwise requires the private keys to be stored in *secure storage* within a trusted platform module (TPM) [FIDa], where storage is sparse and expensive. The main advantage of the *resident key* setting is that it allows for username-less authentication, as the TPM holds a table with user ID (*User Handle*), application domain (*Relying Party ID*) and private key. The downside of *resident key* is that it requires *secure storage* on the user's side for every application the user registered. FIDO defines three communication parties for authentication: authenticator, client and the relying party. The client is an application connecting the authenticator and the relying party, usually a web browser or operating system. The relying party is the application that requires authentication. The authenticator securely stores the user's key material and signs authentication requests. Authenticators communicate only with the client. They do so via CTAP, the Client to Authenticator Protocol, which is a part of the FIDO standard.

[2] The FIDO standard now refers to *resident key* as *"discoverable credential"*. However, as much of the literature and code still refer to *resident key*, we retain that term.

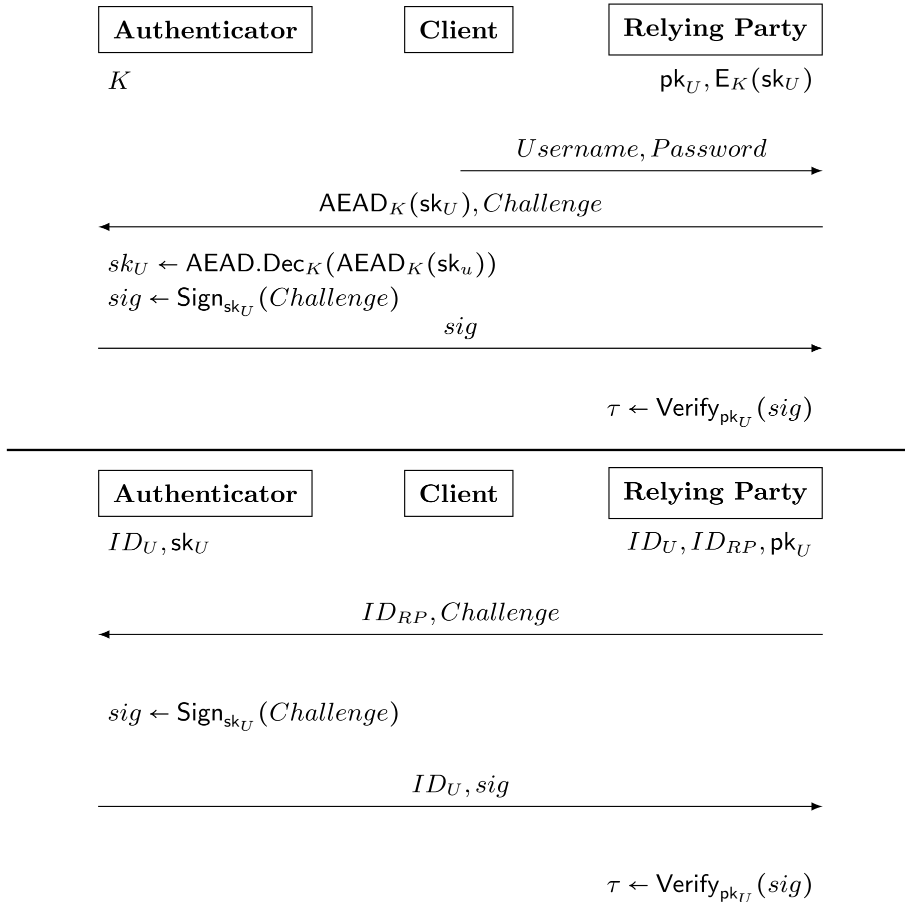


Figure 6.1: Simplified protocol flow diagrams of FIDO instantiations of *non-resident key* (top) and the *resident key* (bottom) authentication. K denotes the symmetric encryption key stored inside the authenticator, sk_U and pk_U the users private and public key, ID_U, ID_{RP} the user's and relying party's IDs. Authentication succeeds if $\tau = 1$.

Table 6.1: Comparison of NIST PQC signature algorithms selected for standardization. The timings refer to the non-optimized, hence portable, reference implementation taken from PQM4 [KRSS] running on a Cortex-M4. The table is divided into security level `I`, `III` and `V`, as defined by NIST. The listed algorithms claim **at least** the security level detailed. SPHINCS+ benchmarks use the SHAKE256 extendable output function (XOF) as hash primitive.

	Size (bytes)			**Runtime (≈Kcycles)**		
`Level I`	privkey	pubkey	signature	keygen	sign	verify
Ed25519[1]	32	32	64	200	240	720
Dilithium2	2 528	1 312	2 420	1 874	7 925	2 063
Falcon-512	1 281	897	666	229 742	62 255	834
SPHINCS+-128	64	32	17 088	50 505	1 182 422	70 501
`Level III`						
Dilithium3	4 000	1 952	3 293	3 205	12 359	3 377
SPHINCS+-192	96	48	35 664	74 890	1 937 690	103 305
`Level V`						
Dilithium5	4 864	2 592	4 595	5 341	15 579	5 610
Falcon-1024	1 281	1793	1280	602 066	136 241	1 678
SPHINCS+-256	128	64	49 856	200 110	4 026 533	108 394

[1]Pre-Quantum algorithm for comparison. Benchmarks taken from Owens et al. [OEKBN+24].

Authenticators can be either roaming (external device) or bound (internal device). Roaming authenticators are allowed to communicate via Bluetooth, NFC or USB.

6.2.2 Post-Quantum Cryptography

Table 6.1 shows the NIST-selected signature algorithms with their claimed security level, key, signatures sizes and performance on a Cortex-M4 embedded processor [KRSS]. The table reveals that post-quantum cryptography is much more expensive than its state-of-the-art pre-quantum counterpart: keys and signatures are much larger and operations require more computational time. Memory limitations can also be problematic for PQC [KRSS], which is of course especially relevant for embedded use cases where bandwidth, storage, memory and CPU time are sparse. A suitable choice in algorithm is therefore integral for FIDO security keys.

Post-Quantum Adoption.

The three standardized PQC signature algorithms rely on different assumptions. Dilithium's [Con21] and Falcon [FHK⁺18]'s security claims are based on the difficulty of solving large instances of structured-lattice problems. These claims are much discussed and sometimes contested in the academic discourse [Ber23, Ber22, Che24]. Because of these debates and the novelty of the algorithms, most implementers, such as Google [Han24], Cloudflare [Wes23], Signal [Kre24] or Apple [Eng24], chose to use lattice-based PQC only in conjunction with a pre-quantum algorithm. This ensures that even if the PQC algorithm contains a major flaw, a pre-quantum attacker will not be able to exploit the system. Combining pre- and post-quantum algorithms into so-called *hybrid constructions* is therefore quite common. However, also *hybrid constructions* are debated. Famously, in 2022 the Natioal Security Agency (NSA) even stated that it *"does not expect to approve"* hybrid constructions for national security citing complexity, interoperability and maintenance concerns [Nat22b].

SPHINCS⁺, on the other hand is *hash based* and its security is solely based on well-understood assumptions of the utilized hash primitive [HK22]. As these assumptions are easier to analyze than their structured-lattice counterparts and since hash-based cryptography has been studied since the early 1970s, it does not seem to suffer from the same trust issues. Exemplary of this is the French Cybersecurity Agency's (ANSSI) position paper on PQC, stating *"any product that includes post-quantum mitigation shall implement hybridation except if the quantum mitigation only relies on hash-based signatures like [...] SPHINCS⁺ [...]"* [ANS24]. The German Federal Office for Information Security (BSI) comes to the same conclusion [fIS24]. The downsides of SPHINCS⁺ are apparent in Table 6.1. Signing is slow and signatures are large, which is especially problematic for embedded use cases.

6.2.3 SPHINCS⁺

Hash-based signature schemes were first described as One Time Signature (OTS) schemes by Lamport in 1979 [Lam79] and further refined by Winternitz the same year [Mer89]. Lamport's scheme relies solely on the properties of the employed one-way function at the expense of being *"one time"*. This means that every private/public key pair can only by used once, as parts of the private key are revealed in the signature. Merkle built up on Lamport's idea by using hash trees to administer multiple OTS public keys under a common root node [Mer89]. This comes at the expense of larger signatures, as the *authentication path* between OTS public key, which is a leaf node, and the root node has to be included. Much more problematic, however, is that a global state has to be kept per key. Maintaining the correct state globally is very challenging and often times impossible, which is why *stateful hash-based signature systems* have mainly seen adoption in niche applications [Lan25]. Using stateful hash-based signature in FIDO is not an option, as the *non-resident key*

authenticator does not hold a state and would have to trust the relying party to not send him a previously used state. In its call for PQC schemes, NIST explicitly called for *stateless* contributions, excluding the otherwise already quantum secure stateful hash based signature schemes.

SPHINCS+ is a *stateless* hash-based signature scheme. It's built mainly on the ideas of the Extended Merkle Signature Scheme (XMSS) [BDH11]. Both rely on the Winternitz One-Time Signature Plus (WOTS+) [Hül13] OTS scheme. Just as XMSS, SPHINCS+ relies on a binary hash tree at its core. However, to become *stateless*, SPHINCS+ needs to have a hash tree so enormous, that choosing a leaf node (private/public key pair) at random is sufficient to exclude any realistic possibility of key reuse. To accomplish this, two tricks are used. First, a so-called *"hypertree"* is utilized. This hypertee contains several layers of XMSS binary hash trees. Each hash tree root within the hypertree is used to authenticate the hypertrees below it. The trees of the lowest hypertee level have key pairs associated to their leaf nodes. This is equivalent to the approach described for multi-tree XMSS in [HRB13]. The novel idea behind SPHINCS+ is to drastically reduce the hypertree's size by authenticating few time signature (FTS) instead of OTS key pairs in its leaf nodes. SPHINCS+ utilizes the Forest of Random Subsets (FORS) FTS for this purpose. Reusing a FORS key pair decreases security only gradually, instead of immediately as in OTS schemes. FORS uses its own tree structure, containing sets of private keys in its leaf nodes for that purpose. As authentication paths shrink due to the much smaller hypertree, using FORS allows for much better signature sizes without impacting security. The exact inner workings of WOTS, XMSS and FORS are detailed in the SPHINCS+ NIST submission [ABB+22]. The subprimitives, WOTS+, hypertree and FORS allow SPHINCS+ to be stateless, but they also offer many options for parameterization. As the SPHINCS+ NIST submission states "*SPHINCS+ can be viewed as a signature template. It is a way to build a signature scheme [...].*".

SPHINCS+ Parameters.

SPHINCS+ instantiations can be fine tuned using six parameters:

- n: The security parameter. Refers to virtually all hash function input, output and tree node sizes in bytes. Commonly used values are 16, 24 and 32 reflecting NIST security levels `I`, `III` and `V`.
- w: The Winternitz parameter specifying how often a message is split during encoding for WOTS+. This does not impact security.
- h: The overall height (layers of nodes) of the hypertree.
- d: Number of layers of subtrees in the hypertree.
- k: Number of trees per FORS public key.
- t: Number of leaves in a FORS tree.

The Winternitz parameter (w) and number of hypertree layers (d) only specify performance tradeoffs. Larger values of w lead to shorter signatures, but more hash function invocations. The number of subtree layers (d) is proportional to the signature size, but inversely proportional to the number of hash function invocation during key generation and signing. All remaining parameters are security relevant. Optimizing these parameters therefore has to be done with caution. The generic quantum security level (bit security) of SPHINCS+ is captured in (6.1) [ABB+22]:

$$b = -\frac{1}{2}\log\left(\frac{1}{2^{8n}} + \sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^{h}}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right) \tag{6.1}$$

The equation ties together all configurable, security-critical parameters with the maximum number of signatures that can securely be produced using the scheme (q) and the number of times an FTS key pair would have to be reused before security is degrading (γ). From the equation it is clear that reducing the maximum allowed number of signatures per SPHINCS+ key pair (q) greatly affects the possibility to further optimize security-relevant parameters. To make equations more concise, we further specify the values $h' = \frac{h}{d}$ as the height of an XMSS tree within the hypertree and l as the number of n-bytes elements in a WOTS+ private key which solely depends on w with $l = \lfloor \frac{\log(\lceil \frac{8n}{log(w)} \rceil (w-1))}{\log(w)} \rfloor + 1$.

SPHINCS+ Runtime.

The aforementioned parameters affect runtime, size and security. More than 90% of the SPHINCS+ runtime is spent inside the hash primitive [KRSS]. Reducing the number of hash-function calls should therefore be a prime objective for optimizing SPHINCS+ instantiations to our use case. The number of hash-function calls for key generation and signing are given in (6.2) and (6.3).

$$\#_{\text{KeyGen}} = 2^{h/d}(lw + l + 2) - 1 \tag{6.2}$$

$$\#_{\text{Sign}} = d\left(2^{h/d}(lw + l + 2) - 1\right) + k(3t - 1) \tag{6.3}$$

It is important to note here, that these equations specify the number of computed hash values. The number of consumed bytes per hash function call, and hence the number of round/compression function invocations are not reflected. These numbers, while performance relevant, depend on the employed hash function and are therefore further discussed in Section 6.4. However, as we will see, the number of computed hash values provides a good enough estimate for runtime.

SPHINCS+ Key and Signature Sizes.

CPU time, bandwidth and storage are a concern when using PQC schemes. SPHINCS+ comes with very competitive key sizes, but large signatures. A

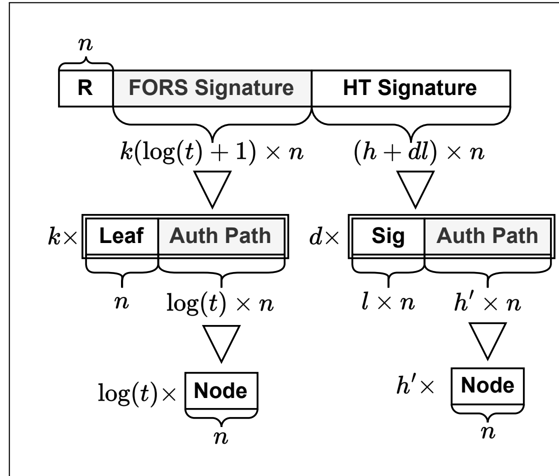


Figure 6.2: Elements and substructures of SPHINCS⁺ signatures with their quantity and size. The value n is in bytes.

Table 6.2: Key and signature sizes for SPHINCS⁺.

	Private Key	Public Key	Signature
Bytes	$4n$	$2n$	$n(h + k(\log t + 1) + dl + 1)$

SPHINCS⁺ public key contains the n-byte hypertree root and an n-byte seed needed for deterministic computation of tree elements. The private key contains the public key as well as an n-byte seed for WOTS and FORS private key generation and an n-byte random value needed for randomization in message hashing. The signature contains a randomness value (n bytes), a FORS signature consisting of FORS leaves with their associated authentications paths ($nk(\log t+1)$ bytes) and an XMSS-like signature containing WOTS+ signatures and corresponding authentication paths ($dn(l+h')$ bytes). Table 6.2 shows the overall length of keys and signatures given these parameters. From the table it becomes apparent that the overall signature size depends on hypertree height (h), number of hypertree layers (d), number of FORS trees (k) and leaves (t) as well as the Winternitz length parameter (l). Figure 6.2 illustrates this fact by visualizing the structure of a SPHINCS⁺ signature.

6.3 Implementation

This section details our implementation of a USB-based CTAP authenticator for use in FIDO, employing PQC signatures in the form of adjusted SPHINCS⁺ signatures. To allow for comparison of results we use the same hardware

and software stack as in the work of Ghinea et al. [GKP+23]. We use the nRF52840 development kit [Norb] with a Cortex-M4F MCU running at 32 MHz. As in [GKP+23], we limit our implementation to 64kB of RAM and keep it generic enough to be easily portable to other platforms. In contrast to Ghinea et al., we also employ the nRF52850's cryptography hardware accelerator CryptoCell 310 [Nora]. However, the CryptoCell is only used for comparison and not required by the implementation. Benchmarks were conducted using the experimental OpenSK [Opeb] security key firmware CTAP2 implementation. OpenSK is based on the TockOS [Toc] embedded operating system. Our SPHINCS+ implementation is based on the C reference implementation. To benchmark within the OpenSK/TockOS environment, we wrote a SPHINCS+ wrapper for the Rust programming language. All major components, such as OpenSK, TockOS and the SPHINCS+ wrapper are written in Rust and published under a permissive license. A repository with benchmarking framework, helper scripts and Rust code is published for reproducibility and to encourage further research. Both *resident* and *non-resident key* scenarios are supported in our implementation. As our SPHINCS+ approach is not a hybrid solution and has very short keys, changes to the OpenSK CTAP implementation are limited to including SPHINCS+ and adding a new algorithm identifier.

6.3.1 Requirements

For comparability we follow the requirements described in [GKP+23]. These requirements stem from the FIDO specification [FIDb]:

- User presence and user verification tokens usually timeout after 30 seconds, but are guaranteed to be valid for at least 10 seconds.
- The size of a CTAP message over USB cannot exceed 7609 bytes.

We therefore aim for commands to finish within 10 seconds and CTAP messages smaller than 7609 bytes. The latter limit stems from the fact that CTAP2 uses a *signed char* (7-bit) value as its length field. Allowing for larger messages (and hence larger signatures) would be a trivial change in the OpenSK firmware. Changing the length field to an *unsigned char* (8 bit) allows for a payload/message size of 15 161 bytes [FIDa], with the remaining bytes being consumed as protocol metadata. For completeness we also include SPHINCS+ instantiations with signatures larger than the CTAP payload maximum in the results, but mark them explicitly as non-compliant. From this we follow priorities as in [GKP+23]:

R1: Key generation must finish in less than 10s.

R2: Key pairs must be smaller than 7 kB.

R3: The private key should be small to allow storing additional credentials.

A1: The login operation is more frequent than registration. Signing should be as fast as possible.

A2: A private key and signature together must fit into a CTAP message.

As [GKP+23], we further limit ourselves to 64kB of RAM and require our implementation to be as portable as the underlying CTAP implementation written in Rust. The NIST-submitted SPHINCS+ parameter sets all fail to achieve A1 and A2. Key generation (R1) and key sizes (R2 & R3) however aren't a problem. Our FIDO adjusted SPHINCS+ instantiations should therefore optimize signing speed and signature size.

Side Channel Resilience.

Just as in [GKP+23], we follow the attacker model described in FIDO's security assumptions [FIDc]. We assume the FIDO client to be trustworthy and acting in the user's interest. This is a necessary requirement for FIDO anyway. Local attacks, such as fault injection [GKPM18] or power side-channel attacks [KGB+18] on SPHINCS+, are therefore out of scope. We only consider timing-based side channels that could be triggered remotely. Here it is important to note that SPHINCS+ is constant time via its construction [ABB+22]. Time-based side channels therefore do not take specific effors to be mitigated.

6.3.2 Adjusting SPHINCS+

Given the lowest acceptable security level `I`, the public and private key are only 32 and 64 bytes large respectively and could be used as a drop-in replacement for elliptic curve keys [Aut] without the need for alterations in e.g. database columns of the relying party. Therefore, only signature size (A2) and signing speed (A1) have to be optimized using Equations (6.1), (6.2) and (6.3). A core variable to tune our SPHINCS+ parameter sets is the number of allowed signatures per SPHINCS+ private key q. Reducing q affects security as defined in (6.1) and further detailed in [ABB+22] and [KP22]. The NIST-submitted parameter sets allow, by NIST specification, for $q = 2^{64}$ signatures per key. Given a user performing one authentication per relying party per day, such a key could be used for far more than a trillion (10^{12}) years, which is clearly overkill. Similar to the approach in [KP22] we therefore lower the maximum amount of allowed signatures to a more realistic level.

Adjusting the Number of allowed Signatures.

In our analysis we allow for $q = 2^8$, $q = 2^{10}$ and $q = 2^{16}$ signatures/authentications. FIDO seems to be a good choice for reducing this number, as every signature has to be approved manually (usually with a tap) by the user. A scenario with real-time constraints or a high volume of signatures can be ruled out. Additionally, FIDO-based authentication

Table 6.3: SPHINCS⁺ key lifetimes based on common session expiration times of popular web services supporting FIDO. Lifetimes are presented in years based on the number of allowed signature q.

q	Google	Sharepoint	Cloudflare	Daily
2^8	9.8	3.5	2.1	0.7
2^{10}	39.3	14	8.4	2.8
2^{16}	2,513.7	897.7	538.7	179.6

is commonly connected to applications with long-running sessions. Google web sessions for example last for 14 days by default [Goo], Cloudflare's for 3 [cfs], Microsoft's for 5 (Sharepoint) to 90 (Entra) days [Mic]. For simplicity we assume a somewhat worst case with respect to the key duration, where a user authenticates to the same relying party every day. Even with the lowest setting of $q = 2^8 = 256$, this user could authenticate using the same key for approximately every workday in a year. After this key expires, a new key would have to be registered. For the user that would mean tapping the authenticator once more during authentication, which is quite cheap. For the best-case scenario with authenticating to Google every 14 days, the same key could be used for around 10 years. With an authentication every day, the settings $q = 2^{10}$ and $q = 2^{16}$ would delay re-registering by around 3 and 180 years respectively. Table 6.3 shows how different settings in q would affect the key lifetimes when used with popular FIDO-ready applications. To store the number of computed signatures, the *"signCount"* variable already included in the FIDO standard [FIDc] can be employed in the implementation.

Adjusting Parameters.

To optimize the parameters discussed in Section 6.2.3 we follow a similar approach to Kölbl et al. [KP22], but optimize for signing speed and signature size instead of verification speed. We employ an explorative approach and built a tool inspired by the SAGE script included in the SPHINCS⁺ NIST submission package[ABB⁺22]. This leads to a multitude of parameters with different tradeoffs further discussed in Section 6.4.1.

Adjusting Hash Functions.

The SPHINCS⁺ NIST submission includes SHA256, SHAKE256 and Haraka instantiations. Haraka is an AES-based construct optimized for use in x86 CPU architectures with AES-NI extension. On our embedded platform Haraka leads to much runtime overhead and was therefore not investigated further. Additionally to SHA256 and SHAKE256, the ASCON hash primitive was used in benchmarks. ASCON is a lightweight hash function scheduled for standardization by NIST. We employed the ASCON C reference implementation. The

CryptoCell 310 hardware accelerator on our evaluation board supports the SHA256 hash function. For comparison, SHA256 was benchmarked in both hard and software.

6.4 Results

In this section we first culminate all previous considerations into adjusted SPHINCS⁺ instantiations for FIDO-friendly values of q. We then benchmark different hash functions relevant to our construction and compare hardware SHA256 to its software version. Next, we present benchmarks to the FIDO-relevant operations *KeyGen* and *Sign* as well as the FIDO operations *MakeCredential* and *GetCredential*. All benchmarks are then discussed and compared to previous work on the same platform.

6.4.1 SPHINCS⁺ Instantiations

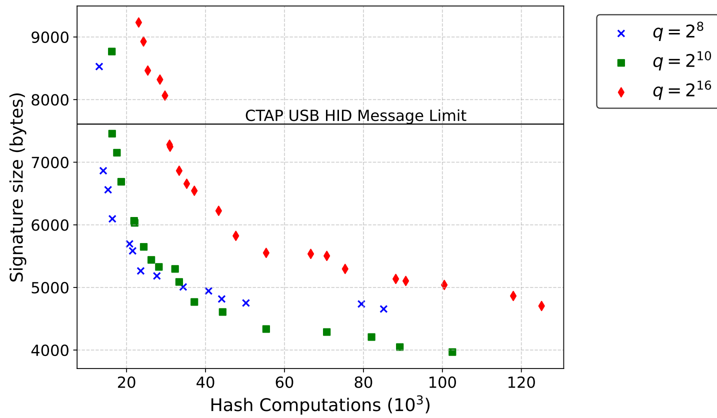
To fulfill the requirements detailed in Section 6.3.1, we utilized our parameter discovery tool. We filtered parameter choices that could lead to a good speed/size tradeoff. The filter starts by sorting all parameter choices by signature size. It then calculates the number of computed hashes and selects a new parameter set if the number of computed hashes is smaller than that of the previously selected. It then outputs a list of all selected parameter sets.

Figure 6.3 shows different parameter choices for all security levels and relevant values of q.

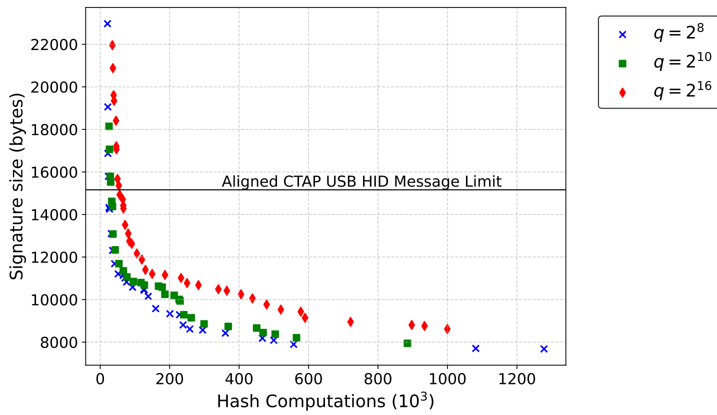
The output shows that level `I` instantiations offer signature sizes lower than the previously described CTAP USB message size limit. For level `III` and `V` there also exist parameter choices with small enough signatures to fit this limit. However, as they require significantly more hash computations and increasing the message size limit is a technical triviality (see Section 6.3.1), the increased message size was set as threshold for them. The value $q = 2^8$ offers only very marginal improvement over $q = 2^{10}$. Using $q = 2^{10}$ therefore seems like a good tradeoff for FIDO. However, we benchmark all relevant choices of $q = \{2^8, 2^{10}, 2^{16}\}$ for completeness. Larger signatures lead to fewer hash calculations but also longer transmission times during FIDO authentication.

6.4.2 Hash Functions

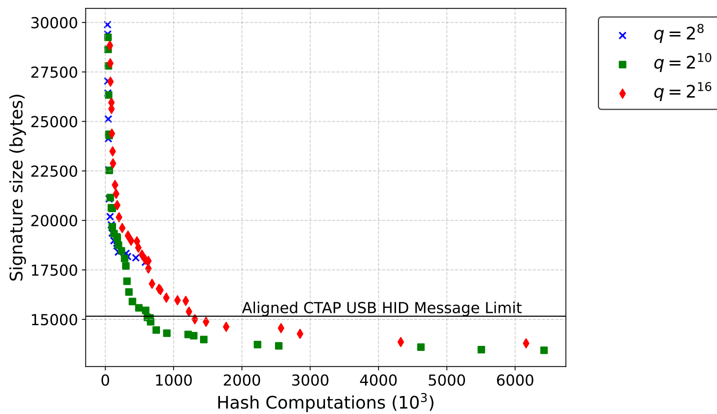
SPHINCS⁺ spends the vast majority of its CPU time inside the hash function. Choosing an appropriate hash function for our use case is therefore integral for performance. SPHINCS⁺, as submitted to NIST, employs the secure SHA256, SHAKE256 and Haraka one way functions. In their work [KSS24], Karl et al. suggest to include the lightweight crypto scheme ASCON [DEMS21] which was recently selected for standardization by NIST. This seems to fit our embedded use case. We therefore focus our performance comparison on SHA256, SHAKE256 and ASCON. Table 6.4 shows the amount of cycles needed on our



(a) Security Level I.



(b) Security Level III.



(c) Security Level V.

Figure 6.3: Signature sizes and number of computed hash values for SPHINCS$^+$ instantiations with a maximum of q signatures per key.

Table 6.4: Cycles spent calculating the minimum amount of hashes required for signing in a given security level. Numbers are given in kilocycles (10^3), averaged over 1000 runs when compiled with *-O3*.

Level	SHA256 (soft)	SHA256 (hard)	ASCON	SHAKE256
I	93 268	130 997	197 534	488 680
III	209 141	293 743	442 940	1 095 792
V	283 664	398 413	600 774	1 486 258

embedded platform to compute the number of hashes required per security level. It comes with no surprise that software SHA256 is much faster than software SHAKE256 on a 32 bit platform. What might be surprising at first sight, is that SHA256 in software also outperforms software ASCON and hardware-backed SHA256 on our platform.

Hardware vs. Software.

However, as Karl. et al. [KSS24] note, not only hardware accelerators themselves, but also the application including data transfer to those accelerators have to be taken into account. In the case of SPHINCS$^+$, a large amount of very small messages (usually a small multiple of n) have to be hashed. This provides a somewhat worst-case scenario for the CryptoCell 310 hardware accelerator, as short messages have to constantly be written into a specific memory segment followed by a relatively slow call to the CryptoCell. This is similar to the observation by van der Lann et al. for SPHINCS$^+$ running on Java Cards [vdLPR$^+$18]. ASCON on the other hand was designed with a very small hardware footprint, not software performance, in mind.

The software version of SHA256 was therefore used for further benchmarks. However, results with other hash functions can easily be extrapolated from the results in Table 6.4. The employed SHA256 implementation is the C implementation taken from the SPHINCS$^+$ submission package. This was done to keep the implementation portable and comes at little cost, as previous work has shown that assembly-optimized SHA256 does not outperform portable code compiled with a modern compiler [KRSS].

6.4.3 Adjusted SPHINCS$^+$ Benchmarks

For FIDO, the runtime of the following four functions is relevant. *KeyGen*: the key generation time in the authenticator (hardware security key). *Sign*: the time needed to sign an authentication request in the authenticator. *MakeCredential*: the time needed to create a new credential (key pair), when requested by the FIDO client. *GetCredential*: the time needed to deliver a signed authentication request to the FIDO client.

Table 6.5: Various SPHINCS$^+$ parameter choices for security level I and $q = 2^{10}$ with their performance results averaged over 1000 runs on an nRF52480 board. Values of t are specified in $\log_2$. All parameters were instantiated with a software-only version of SHA256 and use a Winternitz parameter of $w = 16$.

		Params.				***KeyGen***			***Sign***		
Size (Bytes)	Hashes (#)	h	d	k	t	Runtime (ms) *-Oz*	Runtime (ms) *-O3*	Stack (kB)	Runtime (ms) *-Oz*	Runtime (ms) *-O3*	Stack (kB)
3968	102528	12	2	15	10	3476.1	2332.3	3.3	11385.5	7631.3	14.0
4048	89216	12	2	17	9	3445.0	2320.0	3.3	9378.3	6310.5	14.2
4208	82048	12	2	20	8	3444.1	1161.9	3.3	8349.9	5620.7	14.7
4288	70720	10	2	17	10	1748.6	1176.7	3.3	8548.1	5665.1	14.9
4336	55360	10	2	19	9	1748.1	585.0	3.3	6316.1	4247.7	15.1
4608	44336	12	3	17	9	857.4	585.0	3.2	5037.8	3432.9	15.8
4768	37168	12	3	20	8	866.8	589.7	3.2	4065.8	2724.3	16.3
5088	33328	12	3	25	7	871.0	589.7	3.2	3531.2	2386.6	17.3
5296	32288	8	2	28	8	866.8	293.8	3.2	3790.3	2553.7	17.9
5328	28192	12	4	20	8	432.7	292.5	3.2	3186.4	2156.5	17.9
5440	26264	9	3	25	8	433.0	292.2	3.2	3124.8	2156.5	18.3
5648	24352	12	4	25	7	434.9	292.3	3.2	2650.9	1775.0	18.9
6032	22048	12	4	32	6	439.3	292.2	3.2	2343.4	1553.7	20.0
6064	21912	9	3	33	7	436.9	147.6	3.2	2524.8	1683.1	20.1
6688	18644	10	5	29	7	218.9	147.6	3.2	2150.1	1439.9	21.9
7152	17560	12	6	32	6	221.1	148.4	3.1	1907.3	1264.4	23.3
7456	16340	10	5	40	6	218.9	147.6	3.2	1820.4	1225.5	24.2

As *KeyGen* is a subroutine of *MakeCredential*, and *Sign* of *GetCredential*, their respective runtimes are strongly correlated. However, a parameter set with large signatures might lead to a fast *Sign*, but a slow *GetCredential*, due to transmission latency. For better analysis we therefore benchmarked the four functions individually. The results of *MakeCredential* and *GetCredential* are discussed in the next section. Table 6.5 shows benchmark results of security level I parameter choices for $q = 2^{10}$ that have a small enough signature for use in vanilla FIDO.

Runtimes in the table are captured in milliseconds instead of cycles. This was done to be comparable to previous work, mainly Ghinea et al. [GKP$^+$23]. Estimates of cycle counts can, however, be acquired by multiplying the time with the MCUs frequency, which was configured to 32 768kHz. Unsurprisingly, the number of hashes computed during signing is strongly correlated with the runtime of the signing algorithm. The table also shows that the signature size is inversely proportional to both key generation and signing time. Larger signature sizes also lead to more stack space consumed. Stack space could be drastically reduced, as SPHINCS$^+$ signatures are computed front to back and

Table 6.6: Performance of tailored SPHINCS+ instantiations and the Dilithium-based hybrid construction from [GKP+23] on the nRF52840 board. Both implementations are compiled with *-O3*. Runtimes are averaged over 1000 runs.

	Sizes (bytes)			Runtime (ms)	
`Level I`	privkey	pubkey	signature	keygen	sign
Dilithium2-Hybrid	32	1 344	2 420	197.5	1 320.5
SPHINCS+-129-2^8	64	32	6 864	148.3	1 079.6
SPHINCS+-129-2^{10}	64	32	7 456	147.6	1 225.5
SPHINCS+-130-2^{16}	64	32	7 280	292.3	2 261.8
`Level III`					
Dilithium3-Hybrid	32	1 984	3 293	245.6	2 298.4
SPHINCS+-193-2^8	96	48	14 256	147.5	1 735.6
SPHINCS+-193-2^{10}	96	48	13 080	292.3	2 169.8
SPHINCS+-195-2^{16}	96	48	14 928	292.3	3 410.8
`Level V`					
Dilithium5-Hybrid	32	2 624	4 595	345.3	2 797.2
SPHINCS+-258-2^8 †	128	64	25 120	296.2	2 075.7
SPHINCS+-258-2^{10} †	128	64	24 352	292.1	2 638.0
SPHINCS+-258-2^{16} †	128	64	28 832	292.0	3 216.6

†: Instantiation's signature too large for requirements.

can be streamed [GHK+21]. However, since the consumed stack space is well within the requirements, this is not needed. Compiling the code with *-O3* leads to much improved runtimes, which is consistent with previous work [KRSS]. The largest signature listed (7456B) still fits into the technical context, complies with RAM/stack requirements and adheres to the requirements $A1$ and $A2$ previously defined. We therefore focus on the largest signature still fitting into the technical context.

Table 6.6 shows a direct comparison between the SPHINCS+ results and the Dilithium-based hybrid approach from [GKP+23]. In the table, SPHINCS+ instantiations are denoted as SPHINCS+-b-q, with b being the quantum bit security as defined above. The Dilithium private keys are small, as the implementation stores only a seed value and computes the private key on every signing operation to save expensive TPM storage. Table 6.6 shows that SPHINCS+ levels `I` and `III` with $q = \{2^8, 2^{10}\}$ outperform the lattice-based approach on the board in terms of signing speed at the cost of larger signatures. In security level `V`, SPHINCS+ has competitive speed but would require a relaxation in signature-size constraints.

Table 6.7: Runtime benchmarks in milliseconds of the *MakeCredential* and *GetCredential* CTAP commands in the *non-resident key* setting, averaged over 1000 iterations. Standard deviation of *GetCredential* runtime denoted as σ.

	Make	***Get***	
`Level I`	duration	duration	σ
Dilithium2-Hybrid	547.3	1 808.7	933.8
SPHINCS⁺-129-2^8	283.4	1 595.9	0.2
SPHINCS⁺-129-2^{10}	283.7	1 791.8	0.1
SPHINCS⁺-130-2^{16}	429.2	2 687.9	0.1
`Level III`			
Dilithium3-Hybrid	668.3	2 861.1	1 865.2
SPHINCS⁺-193-2^8	283.2	2 099.9	0.2
SPHINCS⁺-193-2^{10}	429.6	2 751.8	0.2
SPHINCS⁺-195-2^{16}	426.3	3 967.3	0.1
`Level V`			
Dilithium5-Hybrid	799.2	4029.2	2 437.4
SPHINCS⁺-258-2^8 †	425.7	2 803.9	0.2
SPHINCS⁺-258-2^{10} †	429.4	3 406.0	0.1
SPHINCS⁺-258-2^{16} †	432.2	4 075.8	0.1

†: Instantiation's signature too large for requirements.

6.4.4 FIDO Benchmarks

Finally, we conclude with benchmarks on the FIDO-relevant CTAP commands *MakeCredential* and *GetCredential*. All benchmarks were conducted via USB transport. *MakeCredential* generates a new key and sends both public and (encrypted) private key to the client. This is the *non-resident key* setting as introduced in Section 6.2.1. *GetCredential* on the other hand signs an authentication request and returns a signature to the client. Table 6.7 shows the results of these benchmarks. Runtimes for the hybrid construction were reproduced using the code from [GKP⁺23]. A first conclusion is that all previously defined requirements are met for security level `I` and `III`. All CTAP commands finish within 10 seconds and execute well within the RAM requirements. The SPHINCS⁺-based aproach even outperforms the lattice-based, hybrid approach from [GKP⁺23] in these levels. With a relaxation of the maximum message size, our SPHINCS⁺ solution could also outperform their approach in the highest security level.

6.4.5 Discussion and Future Work

A reason for the good performance of SPHINCS+ in this setting could be the required portability. MCU-specific assembly-level optimizations are not possible when aiming for portable code, which is often times a requirement. Dilithium seems to suffer from this portability requirement much more than SPHINCS+. Another reason why Dilithium-backed *GetCredential* commands are slower on average is the command's very long tail. Dilithium's signing operation has a retry loop that discards insecure parameters. This makes its runtime nondeterministic. Ghinea et al. even state that 3% of their Dilithium5 *GetCredential* commands fail to complete within the 10 second requirement due to this retry loop. SPHINCS+ does not suffer from this, as it is constant time by design and the runtimes showed virtually no variance.

Much of the current literature seems to focus on optimizing hardware implementations for SPHINCS+/SLH-DSA [KSS24, Saa24b]. While these hardware-backed systems promise impressive performance improvements, they might be able to perform even better when allowing (or mandating) custom SPHINCS+ instantiations. Comparing and combining adjusted SPHINCS+ parameter sets with custom hardware seems like an interesting avenue for future research.

6.5 Conclusion

In this chapter we proposed a practical, post-quantum solution for hardware-security-key-based authentication using tailored stateless hash-based signatures. The results show that adjusting SPHINCS+ to specific embedded applications that require signing can yield very competitive results that rely on only few, well-understood assumptions. Especially in an embedded setting, where resources are sparse this opens many possibilities. Furthermore, our solution is highly portable and reduces complexity as it does not require sophisticated implementation tricks and can be used without a hybrid construction. These results emphasize that SPHINCS+ should not be seen as a signature scheme, but as a very flexible framework for creating signature schemes.

We implemented and benchmarked our solution with various parameterizations and showed that it can even outperform lattice-based solutions from the literature. Our implementation and benchmarking framework was published to encourage further research in this direction.

Part III

Supplementary Works

Chapter 7

Security Analysis of the Olvid Secure Messenger

This chapter is based on the following unpublished paper. The paper was submitted to the ACM Conference on Computer and Communications Security to be held in November 2026 and is currently under review.

> Noemi Terzo, Cas Cremers, Ruben Gonzalez, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. Security analysis of the olvid secure messenger. *unpublished*

In it, we present an analysis of the cryptographic core of the French encrypted messaging app ***Olvid***. It is, among other things, a recommended instant messaging solution for french government officials. We employ the TAMARIN prover for a formal analysis of Olvid's authenticated key-exchange and continuous-key-agreement protocols, and conduct an analysis of the Java implementation targeting Android devices. Even though the formal analysis confirms that the Olvid protocols achieve security properties such as mutual authentication, session-key secrecy, forward secrecy, and replay protection, the analysis also reveals that the Olvid protocols fail to achieve security under certain key-reveal patterns. Furthermore, source-code analysis reveals at least one timing leakage vulnerability, for which we present a practical key-recovery attack. To enhance Olvid's security and facilitate future analyses, we propose design changes aimed at mitigating the identified vulnerabilities.

Contribution. In this chapter we present the first extensive security analysis of the cryptographic core of Olvid consisting of a systematic formal analysis and a review of the Java client source code with a focus on implementation security. As a result we present the first formal analysis model of Olvid's cryptographic core using the TAMARIN symbolic protocol prover. Furthermore, we demonstrate practical attacks on the Olvid implementation and publish their Proof-Of-Concept code.

Associated Software. The software as well as TAMARIN models associated with this chapter are available under `http://doi.org/10.5281/zenodo.17381244`. Included is a Docker image for the TAMARIN protocol models and proofs. Further, it contains the source code for replicating the timing side-channel attack. Two included README files provide detailed instructions on how to run the experiments and reproduce the results.

Organization of this chapter. The chapter starts with an introduction in Section 7.1 that gives background on Olvid's relevance and technology stack. It further details an extensive list of this chapter's contributions. In Section 7.2 we elaborate on the preliminaries, such as employed security models, needed in the following sections. Section 7.3 builds up on this by detailing the relevant inner workings of the Olvid messaging application. We then continue with a formal analysis in the TAMARIN symbolic protocol prover [MSCB13] in Section 7.4 and the description of multiple implementation flaws together with an assessment of their impact in Section 7.5. Section 7.6 concludes the chapter with a reflection on the results and potential avenues for future work.

7.1 Introduction

The Olvid secure messenger was released in 2019, and like other messengers, it promises state-of-the-art end-to-end encryption, but its promises in terms of security go significantly beyond this. Not only does Olvid assume an untrusted server in the sense that is typically assumed for end-to-end encrypted communication, but also the Olvid website (as of April 12, 2025) claims that *"no third party could ever identify the participants, not even the server"* that there is *"no trace of any metadata"*, while at the same time *"users identity [is] guaranteed without the need for a trusted third party"* [Olv25a]. The lead developers of Olvid, Finiasz and Baignères, on their private websites describe Olvid as *"the most secure professional messaging application ever made"*.[1]

Applications with bold security and privacy claims are not rare, and very often such claims do not hold up under closer inspection by security experts [ABCP24, AMPS22, ACDJ23, MRAP23]. However, there are multiple reasons to believe that Olvid might be different. The lead developers are both established cryptographers who worked in leading European research groups in cryptography before joining Olvid. Also, in 2020 and 2021, Olvid received a level-one security certification ("certification de sécurité de premier niveau", CSPN) of the French government agency ANSSI. The code of the Olvid client is open source and the reports pertaining to the ANSSI certification are publicly available on the Olvid website. In 2023, Olvid hit the news when then French Prime Minister Élisabeth Borne urged employees of the government to stop using WhatsApp, Signal, or Telegram, and instead migrate to Olvid. In this context, then digital minister of France Jean-Noël Barrot called it the *"most secure instant messenger in the world"* [Bar23b]. As a result, Olvid is

[1] See `https://finiasz.net/` and `https://www.baigneres.net/`.

now used by employees of the Élysée and multiple ministries in France. In total, Olvid has about 200 000 active users, mostly from France, with usage in other European countries steadily increasing [BF25].

Despite this endorsement of Olvid and its deployment for sensitive communication, very little effort has so far gone into public security analysis of Olvid. The only independent public analyses of Olvid's security that we are aware of are the reports by Synacktiv in the context of the ANSSI certification [Syn20, ANS20, Syn21], an analysis by Abdalla of Olvid's trust-establishment sub-protocol [Abd20], and an investigation of the post-compromise-security properties of Olvid and other messengers by Cremers, Fairoze, Kiesl, and Naska [CFKN20].

This is somewhat surprising also because unlike many other messenger services including WhatsApp, Skype, Facebook Messenger, and of course Signal, Olvid does not use the Signal protocol, which is thoroughly studied and widely regarded as the "gold standard" for (continuous) key agreement in secure messaging [ACD19, CGCD+17, BJKS24, FG24, FJ24]. Instead it uses its own suite of protocols for this *cryptographic core*: The Channel Creation with Contact Device (CCCD) Protocol plays roughly the role that X3DH plays in the Signal protocol, i.e., it is used to agree on symmetric keys between two parties that do not yet have a secure communication channel. These symmetric keys are subsequently used in an Oblivious Channel to exchange payload data and continuously updated through a (symmetric) self ratchet and the (asymmetric) Full Ratchet Protocol. The combination of these two ratchets is akin to Signal's double ratchet.

Contributions. In this chapter we present the first extensive security analysis of the cryptographic core of Olvid consisting of a systematic formal analysis and a review of the Java client source code with a focus on implementation security.

- For the formal analysis we model the cryptographic core of Olvid in the Tamarin symbolic protocol prover [MSCB13]. This model is partly derived from the technical specification of Olvid [BF24], but for many aspects not covered by this specification, we consulted the Olvid source code.
- Building on this model, we present computer-verified symbolic proofs of multiple properties that one typically expects from authenticated key-exchange and continuous key-agreement protocols, such as session-key secrecy, mutual authentication, forward secrecy, and replay protection.
- We furthermore investigate if Olvid's cryptographic core also meets stronger security properties similar to Signal. We show that the Full Ratchet Protocol in principle offers a form of Post-Compromise Security (PCS) for a single ratchet chain, similar to Signal[2].

[2]Note however that already [CFKN20] showed that protocol-level PCS security does not extend to conversation level, either in Signal, or in Olvid.

We next investigate whether Olvid's cryptographic core also achieves key secrecy after certain combinations of ephemeral and static keys have been compromised. This would be mandated by, e.g., the well established extended Canetti-Krawczyk (eCK) model for authenticated key exchange [LLM07]. Our analysis shows that—unlike for Signal—this is not the case for some key-reveal patterns and is the case for other patterns only under the non-standard assumption that ephemeral public keys do not leak to the adversary.

- In our source-code analysis we found that Olvid is at several spots assuming certain values to be in a canonical representation without actually enforcing that this is the case. As a result of this, signatures do not achieve the strong unforgeability notion. We were, however, not able to escalate this to a full attack.

- We furthermore identify a rather slow, but persistent denial-of-service attack vector which stems from the way Olvid implements replay protection.

- Most notably, our code analysis also revealed a timing leakage in the implementation of elliptic-curve scalar multiplication and present a proof-of-concept exploit on an x86_64 CPU showing that this leakage is in principle exploitable.

We conclude the chapter by discussing various aspects of the design of Olvid that we believe should be improved to ease further security analysis, and improve transparency towards the users, in particular regarding the anonymity properties.

Related Work. There is a large body of literature analyzing and formalizing real-world security protocols; see, e.g., [PST23, RMS18, CGCD⁺17, AJM22, ACD19, Ste24] for some examples. During the last decade, formal symbolic analysis has emerged as a technology that can aid analyzing such complex systems, both for attack-finding and establishing security guarantees, notably using the TAMARIN prover [CHSvdM16, CHH⁺17, BDH⁺18, BSTP21, MSCB13, LSB24, CHSW22]. Part of our work falls into the latter category, and part of our analysis is manual.

There is currently only one public technical report presenting formal-analysis results for Olvid [Abd20]. The report is very limited in scope and restricted to the short-authentication-strings (SAS) mechanism that is part of Olvid. The core conclusion is that the Olvid-specific changes to the SAS subprotocol do not invalidate its security arguments, building on Pasini [Pas09]. The certification reports by Synacktiv and ANSSI [Syn20, ANS20, Syn21] only give a high-level summary of the protocols used in Olvid, but do not provide any formal analysis of these protocols.

Post-Compromise Security (PCS) [CGCG16] is a form of security that may be achieved after a device has been compromised, notably if the adversary

is passive for a short period after compromise, which may allow the compromised party to "heal" by exchanging new ephemeral keys to bootstrap future secure communications. PCS has been extensively studied in the context of secure messaging, notably for Signal [CGCD+17] and in general, e.g., [BBL+23]. While many subtle variants of PCS have been proposed, it has remained elusive in practice: [CMN24, CFKN20] show that real-world messaging apps, including Olvid, do not achieve PCS in black-box experiments. Moreover, [CJN22] shows that applications that need to be resilient to loss of dynamic state fundamentally cannot achieve PCS.

A large body of works exploits timing variations in cryptographic implementations. See [LZJZ22] for a recent survey. Since its introduction, `Flush+Reload` [YF14] has been a popular attack technique for creating and observing timing differences [IAIES14, PGBY16, GSM15, RPS18], including for attacks on elliptic curves [FWC16, BvdPSY14]. While the Montgomery ladder implementation [Mon87] used in Olvid is considered more resilient to timing attacks than others [JY02, OKS00], past work has demonstrated that it is not always secure [GVY17, GPTY18]

7.2 Preliminaries

This section introduces the additional background required for the following chapters.

7.2.1 Security models for Key Exchange

There is a long history of security models for key exchange of ever-increasing strength. Classical notions of key exchange security, such as [BR93] consider an active adversary that controls the network, and which can reveal selected session keys. The security notion requires the session key of a specific session (often referred to as the Test session) to be indistinguishable from random, even if the adversary can learn the session keys of other sessions. Many subtleties in this domain then arise from the exact definition of "other session": one might expect this to be any session that computes a different session key. However, in most security models for key exchange, the adversary can also reveal the session key of sessions whose variables/messages differ from the Test session. Effectively, this models a form of implicit authentication.

Later security models for key exchange give the adversary the ability to reveal long-term keys of some participants. This allows for modeling perfect forward secrecy and identity misbinding attacks, by considering that not all participants might be honest, or could be compromised in the future.

Finally, newer key exchange models such as CK [CK01] and eCK [LLM07] also consider an adversary that can compromise the randomness of specific sessions.

7.2.2 Security models for CKA

While standard models for key exchange consider stateless protocols that start from asymmetric key pairs, and produce a shared session key, the notion of CKA (*continuous* key agreement) [ACD19] starts from a shared symmetric key, and produces a new shared key. This primitive was introduced in [ACD19] as part of an attempt to decompose the Signal protocol design into building blocks. The corresponding security model is relatively simple, as the adversary is assumed to be passive.

7.2.3 The TAMARIN prover

For our formal analysis, we use the TAMARIN prover [SMCB12], which is a state-of-the-art tool for the automated analysis of security protocols. TAMARIN performs symbolic analysis based on the Dolev-Yao model, where cryptographic primitives are treated as "perfect" black boxes. Cryptographic primitives and their properties are modeled using a term algebra with an equational theory. TAMARIN takes the protocol to analyze, the adversarial capabilities that define the threat model, the security properties and verifies the properties or provides counterexample traces that constitute attacks to the protocol when it is running in the specified adversarial environment. Both the protocol and the adversary are defined using *multiset rewriting rules*, which induce a labeled transition system. A *rule* has a premise and a conclusion, which in turn are multisets of so-called facts:

```
1 [ PremiseFact1(.), PremiseFact2(.), ... ]
2 --[ ActionFact1(.), ActionFact2(.), ... ]->
3 [ ConsequenceFact1(.), ConsequenceFact2(.), ... ]
```

Rules modify the system state by adding or removing facts. A *fact* is a user-defined symbol *FactName*(*t1*, *t2*, ..., *tn*) that contains *terms ti* as arguments. Facts can be linear, meaning they are consumed upon rule execution, or persistent (denoted by a ! prefixed to the fact name), remaining in the system indefinitely. Each concrete execution of the transition system is a sequence of states with transitions labeled with rule instances. For each execution, the corresponding sequence of action facts is called a *trace*.

Security properties are expressed as first-order logic formulas over traces, referring to the actions of the protocol rules and quantified on timepoints and terms. Timepoint variables are prefixed by the # symbol. To refine the analysis by filtering out traces that do not satisfy specific conditions, it is possible to utilize *restrictions*, which are modeled in the same way of the security properties and allow TAMARIN to focus on relevant traces. Within this transition system, the protocol and adversary interact by exchanging messages over the adversary-controlled network.

7.2.4 Cache side channel attacks

Modern processors employ an array of techniques aimed at bridging the speed gap between the fast execution core and the slower memory. One of the main technique is the use of caches. These are small and fast memories that store recently and frequently accessed data, exploiting the spatial and temporal locality of the software. Multiple works observed that sharing the use of caches between multiple workloads results in side-channels that leak program behavior across security boundaries [OST06, Per05, YF14]. Over the past two decades, several techniques for mounting cache attacks have been proposed, demonstrating leaks from cryptographic code [OST06, Per05, YF14], user activity [GSM15, SKH+19], and other software [BMD+17, SSD21, YFT20].

In this chapter we focus on the `Flush+Reload` technique [YF14], which consists of two steps. In the first step the adversary uses a cache-management instruction to evict a target cache line from the cache. The adversary then waits for a short while, to give the victim time to execute. Then, in the second step, the adversary measures the time it takes to access the previously evicted line. Fast access indicates that the line is cached, presumably due to victim access. By repeating these steps, the adversary can monitor the victim's access patterns to the monitored cache line.

Traditionally, `Flush+Reload`-based side-channel attacks were only demonstrated on Intel processors. However, in recent work, Lin et al. [LWL+24] demonstrate an attack on an Arm processor. Specifically, their work demonstrates a new technique for mapping application code across security domains, exposing apps to attacks such as `Flush+Reload` that exploit memory sharing.

7.3 The Olvid Encrypted Messenger

The Olvid messenger targets a similar user base as other prominent secure messengers, such as Signal. It offers end-to-end encrypted communication and group communication features, allowing users to securely exchange messages with others. Additionally, Olvid supports multiple devices per user. One notable difference between Olvid and other secure messengers like Signal is the registration process. While Signal requires users to provide a phone number to verify their identity, in Olvid users are only required to provide basic information, such as their name and surname, with all other details being optional. Upon creating an account, the Olvid app generates the necessary cryptographic keys on the client-side. This includes the long-term KEM key pair required for encrypting messages sent via the Asymmetric Channel, as well as the signing-verification key pair. The Asymmetric Channel is a core component of the Olvid messenger, and its description will be provided in subsection 7.3.2. Once a user has created an account and generated their associated keys, to establish a contact relationship with another user they have to go through the authentication process. Unlike Signal, the free version of Olvid requires users to add contacts through out-of-band channels and to run the SAS-based authentication proto-

col [PV06]. This approach ensures that users can securely authenticate with each other without relying on a centralized directory.

The SAS-based authentication protocol used by Olvid relies on the sharing of cryptographic tokens via QR codes or URLs, which serve as unique identifiers for secure peer authentication. These identifiers consist of a URL to the Olvid backend server, as well as the verification key and long-term KEM public key. Importantly, these identifiers do not include any user-related metadata, such as usernames or phone numbers. In contrast, the paid enterprise version of Olvid uses a centralized contact discovery system based on the open-source Keycloak identity and access management solution [Olv25b]. This approach provides a more traditional contact discovery experience, where users can search for and connect with other users in a centralized directory. Once users have authenticated with each other, they can establish a secure communication channel, named Oblivious Channel, using the Channel Creation with Contact Device Protocol (CCCD). This protocol is an authenticated key exchange protocol that ensures the establishment of the keys used in the Oblivious Channel and is also run every time a user associates a new device to their account. The Oblivious Channel is based on a ratchet mechanism similar to the Signal's double ratchet and is used to encrypt all application messages between users, as well as protocol messages exchanged between users' instances of Olvid. All messages exchanged between two users are sent encrypted through the Olvid infrastructure, which is hosted on Amazon Web Services (AWS). The client-server communications are secured using TLS 1.3, with TLS 1.2 also supported as a fallback. To renew the root keys used in the Oblivious Channel, the Full Ratchet Protocol (FRP) is run.

In this chapter, we focus on the cryptographic core of Olvid, which assumes that the out-of-band exchange of public-key material and authentication has successfully completed. This assumption is justified by the prior analysis of the SAS-based protocol by Abdalla [Abd20]. Specifically, we assume that Alice has Bob's authentic public-key package containing his verification key and long-term KEM public key, and vice versa.

It is worth noting that Olvid's terminology deviates from the conventional cryptographic meaning. For example, the Oblivious Channel does not refer to concepts such as Oblivious RAM (ORAM) or Oblivious Transfer (OT), and the ratchet mechanism does not follow the Signal protocol exactly. Additionally, the Asymmetric Channel and Oblivious Channel are not traditional cryptographic protocols, but rather protocols used to encrypt and authenticate message keys via an Authenticated Encryption scheme. The Asymmetric Channel uses a KEM-based hybrid encryption scheme to encrypt messages to the public key of the receiver, while the Oblivious Channel uses symmetric keys that evolve through two ratchet protocols. Both the CCCD and FRP protocols output seeds for the ratchet rather than the final symmetric keys themselves.

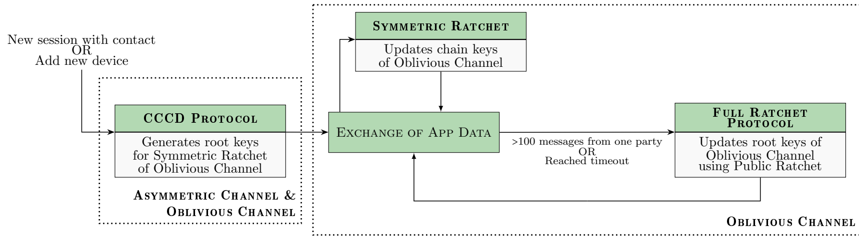


Figure 7.1: High-level view of the Cryptographic Core of Olvid

7.3.1 Threat Model

Secure messaging systems are typically evaluated against an active network adversary with extensive capabilities. This adversary is capable of intercepting, reading, modifying, reordering, replaying, and injecting messages, as well as impersonating users by sending new messages to any participant in the system. Additionally, the model must account for the possibility of malicious insiders, meaning that some users within the system may act dishonestly or attempt to subvert security guarantees. Furthermore, the server is assumed to be untrusted, meaning it could be compromised, malfunction, or collude with an adversary to extract metadata or manipulate communications. In such a scenario, a meticulously designed secure messaging system should prioritize minimizing reliance on the server for security-critical operations and ensure confidentiality, integrity, and authenticity solely through cryptographic mechanisms. Olvid explicitly claims to provide post-compromise security and forward secrecy, which suggests that its cryptographic design must be resilient against key compromise at some point in time. This includes protection against compromised randomness, ensuring that even if an adversary gains temporary access to encryption keys, they cannot decrypt past or future messages. However, since Olvid does not provide a formal threat model in its documentation or white papers, we will use the threat model above, which aligns with existing security research, to analyze its security guarantees.

7.3.2 The cryptographic core of Olvid

The starting point for the cryptographic core of Olvid as considered here is that two users **A** and **B** have obtained each other's authentic long-term key packages containing KEM public keys and signature verification keys. Hence, **A** starts with state

$$\gamma_A = (kempk_A,\ kemsk_A, signpk_A,\ signsk_A, kempk_B,\ signpk_B)$$

and **B** starts with state

$$\gamma_B = (kempk_B,\ kemsk_B, signpk_B,\ signsk_B, kempk_A,\ signpk_A).$$

High-level view. Olvid uses two main protocols, the Channel Creation with Contact Device (CCCD) Protocol and the Full Ratchet Protocol, and two sub-protocols, Asymmetric Channel and Oblivious Channel. To bootstrap, parties **A** and **B** first execute the CCCD Protocol, which is an authenticated key exchange, and agree on shared symmetric-key seeds s_A and s_B. The CCCD Protocol uses Asymmetric Channel as a subroutine in its first part; in its second part, once shared secrets are established, it performs key-confirmation that uses Oblivious Channel as subroutine. The symmetric key seeds s_A and s_B are subsequently used and updated by Oblivious Channel for payload encryption; s_A is used and updated for messages sent from **A** to **B**; s_B is used and updated for messages sent from **B** to **A**. Informally, Oblivious Channel includes a symmetric ratchet. Furthermore, when at least 100 messages have been sent by a party or when a timeout of one week is reached since the last execution of the Full Ratchet Protocol, the Full Ratchet Protocol is executed to derive a new key seed for the Oblivious Channel, and can be informally regarded as an asymmetric ratchet[3]. Figure 7.1 depicts this high-level view as a diagram.

Asymmetric Channel. Olvid's Asymmetric Channel is a public-key encryption (PKE) scheme built from a KEM and an authenticated encryption (AE) scheme. The KEM is instantiated with the ECIES-KEM construction [Sho01], a Diffie-Hellman-based KEM that is IND-CCA-secure. For the Diffie-Hellman routine Olvid uses Curve25519 [Ber06b] but the source code also includes an implementation using the Million Dollar Curve (MDC) [BDF+15]. For authenticated encryption Olvid adopts the Encrypt-then-MAC approach, using AES-256-CTR combined with HMAC-SHA-256.

The Olvid specification describes this PKE scheme as a unidirectional communication channel, so we use corresponding notation in Figure 7.2. Note that Asymmetric Channel is not following the KEM/DEM construction from [Sho01]: rather than using k_1 to encrypt the message m, it first encrypts the message key mk_A with k_1 and then encrypts m with that message key.

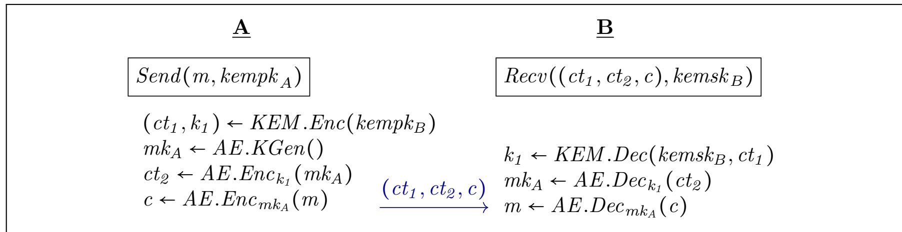


Figure 7.2: The Asymmetric Channel protocol

When Asymmetric Channel is used as a subroutine of higher-level protocols, we will use notation

[3]These are the thresholds given in the Olvid specification [BF24]; the Android implementation in the analyzed version 3.6 uses 500 messages and 24h as thresholds.

$$\xrightarrow[\text{Asymmetric Channel}]{m}$$

to mean that party **A** executes $Send(m, kempk_B)$ to encrypt message m and send the ciphertext c to **B**, followed by party **B** executing $Recv(c, kemsk_B)$ to receive c and decrypt to m.

Oblivious Channel. Olvid's Oblivious Channel is essentially an authenticated-encryption scheme with symmetric self-ratcheting of the key. Again, authenticated encryption uses AES-256-CTR with HMAC-SHA-256. Additionally, Oblivious Channel uses a PRNG and a KDF, which are instantiated with HMAC_DRGB as specified in [BK25] with SHA-256 as hash function.

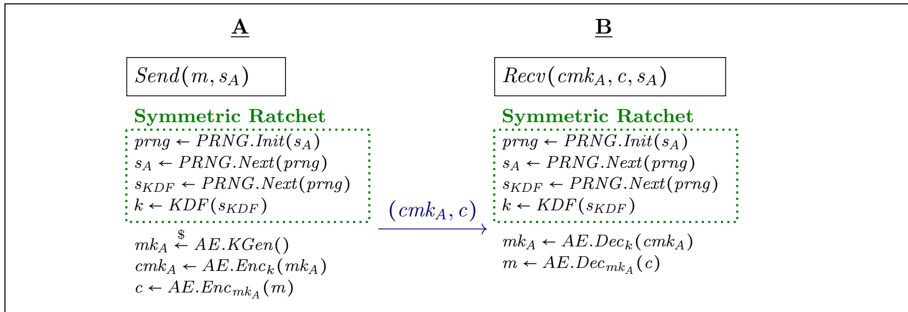


Figure 7.3: The Oblivious Channel protocol with the symmetric-ratchet component marked in green

Just like for the Asymmetric Channel, we use unidirectional-channel notation in Figure 7.3. Note that there are multiple levels of "encryption indirection" in Oblivious Channel. First, the key seed s_A is used as secret input to PRNG to derive an updated key seed s_A and another seed s_{KDF}. This seed is then used as input to KDF to generate a symmetric key k, which is used to encrypt the message key mk_A, which in turn is used to encrypt the message m.

When Oblivious Channel is used as a subroutine of higher-level protocols, we will use notation

$$\xrightarrow[\text{Oblivious Channel}]{m}$$

to mean that party **A** executes $Send(m, s_A)$ to encrypt message m and send the resulting ciphertext c to **B**, followed by party **B** executing $Recv(c, s_B)$ to receive c and decrypt to obtain m.

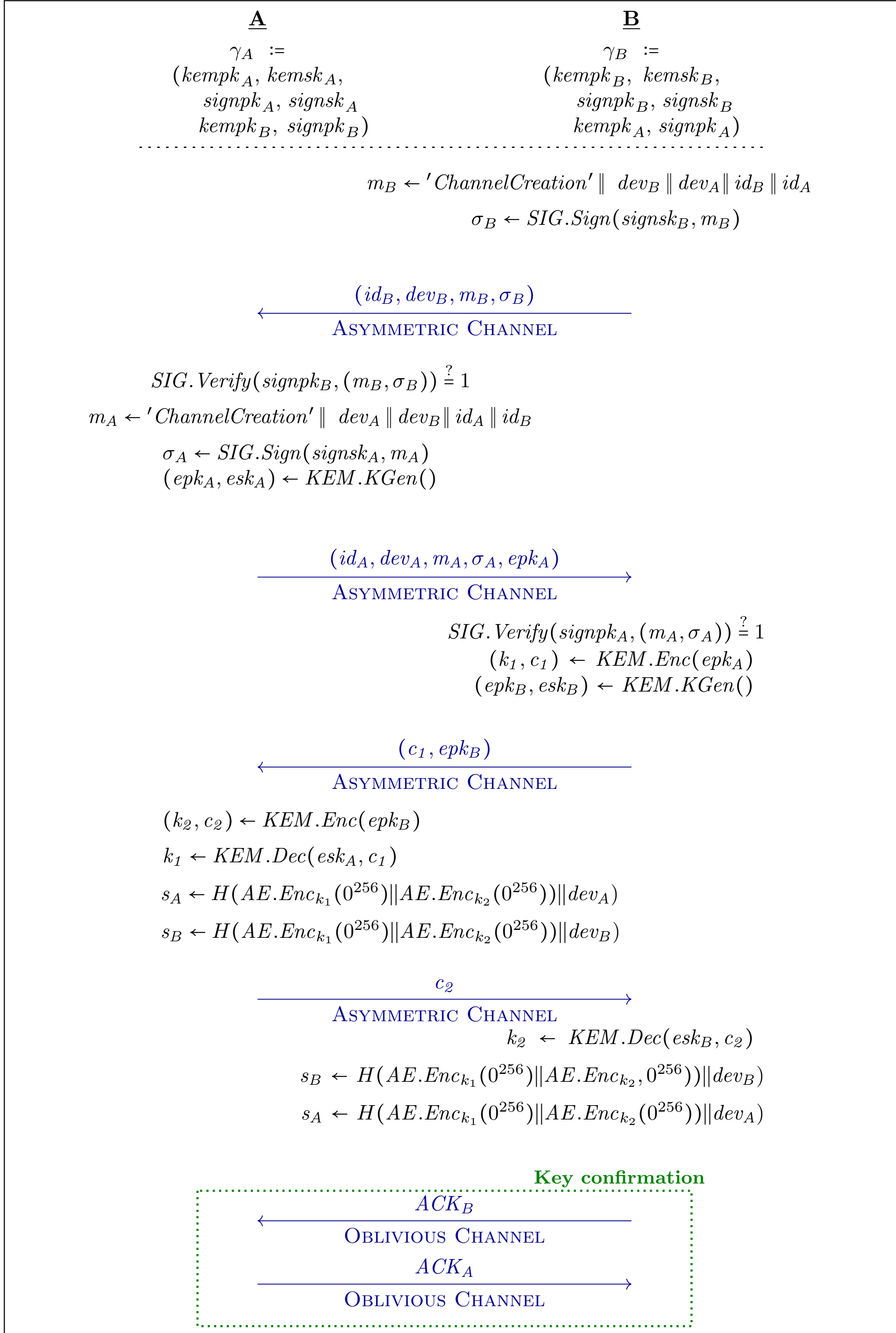


Figure 7.4: High-level overview of the CCCD Protocol

Channel Creation with Contact Device Protocol. This protocol implements an

authenticated key exchange between two parties **A** and **B** that have already obtained each other's authentic long-term KEM and signature keys. The output of the protocol are symmetric-key seeds s_A and s_B. Prior to executing the protocol, party A sends a message to contact B containing A's identity id_A and device identifier dev_A. This message triggers the execution of the CCCD PROTOCOL by B, provided that B has A in its contact list and A is active. The KEM is again instantiated with ECIES-KEM on Curve25519, the signature algorithm SIG is the Edwards-Curve Digital Signature Algorithm (EdDSA) [BDL+11] over the MDC curve. The hash function H used in s_A and s_B is SHA-256. We show a simplified version of the protocol in Figure 7.4. The strings ACK_A and ACK_B exchanged in the key-confirmation phase contain additional information about the participants, for example, an encoding of their profile picture.

Full Ratchet Protocol. The Full Ratchet Protocol exchanges fresh key material through an already established OBLIVIOUS CHANNEL to update the key seed s_A (or s_B if roles are reversed). Together with the symmetric ratchet that is part of OBLIVIOUS CHANNEL, the protocol can be seen as Olvid's version of the double ratchet in Signal. While the symmetric ratchet alone would be sufficient to achieve forward security, the FULL RATCHET PROTOCOL is required for post-compromise security. All cryptographic primitives in the FULL RATCHET PROTOCOL are instantiated as in the CCCD PROTOCOL. We show the protocol in Figure 7.5.

Note that if **A** aborts the protocol after B has already updated s_A this does not cause a desynchronization: **A** will simply reinitiate the FULL RATCHET PROTOCOL from scratch by sending the first message, and **B** will still be able to decrypt it because it retains a set of older seeds.

7.4 Formal analysis using TAMARIN

In this section we first explain how we model in the cryptographic core of Olvid in TAMARIN. We then use this model to prove many security properties and show that other properties of state-of-the-art secure-messaging protocols do not hold.

7.4.1 Formally modeling Olvid

We first remark on our models of the signatures, encryption, and KEM primitives. As the implementation makes use of malleable signatures, we use the malleable signature models defined in [JCCGS19]. These models allow for multiple distinct signatures to be valid for the same message and signing key. For authenticated encryption, we use the built-in encryption model (see [CDJZ23]). For KEMs, we use a variant of the model used in [LSB24], which we adapt for our threat models.

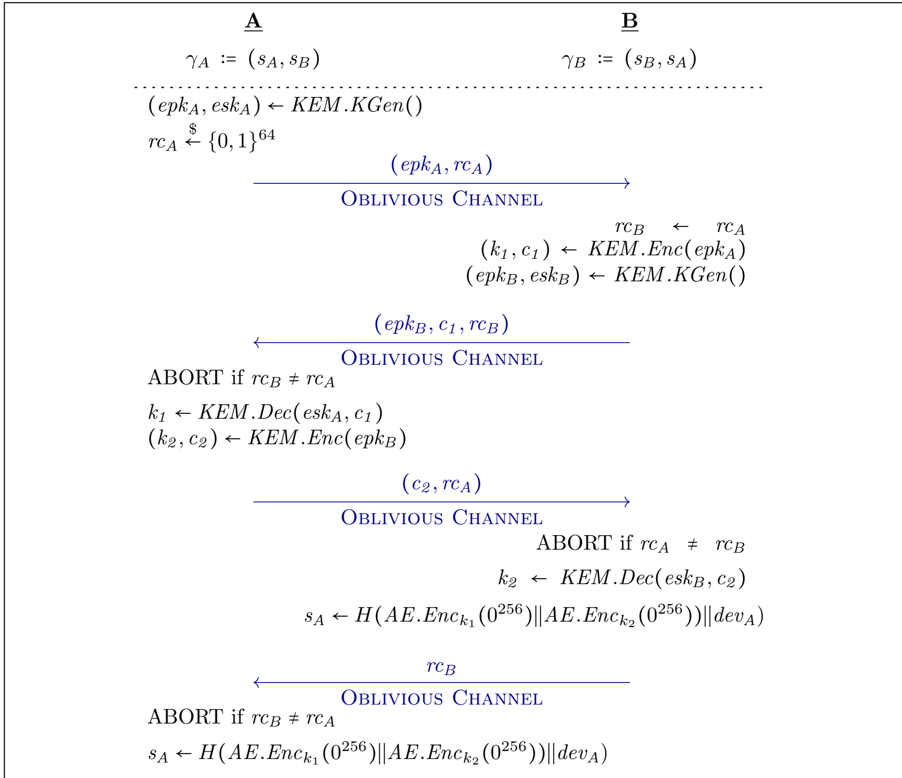


Figure 7.5: High-level view of the Full Ratchet Protocol

Asymmetric Channel. The following macro defines the structure of a record transmitted via the Asymmetric Channel:

```
1 AsymChanRecord(actor, peer, devActor, devPeer, kemPkPeer, k, rand, msgKey,
      msg) = <<actor, peer, devActor, devPeer, <encaps(k, kemPkPeer, rand),
      senc(msgKey, k)>>, senc(msg, msgKey)>
```

When a party *actor* sends a message *msg* to the party *peer* through the channel established between *devActor* and *devPeer*, it follows the *Send* procedure shown in Figure 7.2. In the macro, consistent with the actual implementation, the record includes additional header fields: *actor*, *peer*, *devActor*, *devPeer*. The component $< encaps(k, kemPkPeer, rand), senc(msgKey, k) >$ corresponds to cmk_A in Figure 7.2. Note that, following the standard notation used in Tamarin, the arguments of the encryption function appear in reversed order compared to conventional representations.

Oblivious Channel. Similarly, the macro for the Oblivious Channel is the defined as follows:

```
OblvChanRecord(actor, peer, devActor, devPeer, seedActor, msg, msgKey) =
    <<actor, peer, devActor, devPeer, PRNG(seedActor, '1'), senc(msgKey,
    hkdf(PRNG(seedActor, '2'), 'authEncKey'))>, senc(msg, msgKey)>
```

This models the encryption mechanism in Figure 7.3. The labels *'1'* and *'2'* match the corresponding calls to *PRNG.Next* in the figure. Here, *seedActor* corresponds to s_A before being ratcheted and $hkdf(PRNG(seedActor,'2'),'authEncKey')$ corresponds to k. The symmetric ratcheting of chain key seeds is handled directly within the *ObliviousChannel* fact: when a party A sends a message, its local Oblivious Channel state is updated from *ObliviousChannel(A, B, devA, devB, seedA, seedB, counter)* to *ObliviousChannel(A, B, devA, devB, PRNG(seedA, '1'), seedB, counter+1).* Similarly, when the receiving party B processes the message, its Oblivious Channel state changes from *ObliviousChannel(B, A, devB, devA, seedB, seedA, counter)* to *ObliviousChannel(B, A, devB, devA, seedB, PRNG(seedA, '1'), counter).* This mechanism ensures that seed values are ratcheted in synchrony with message transmission and reception, while maintaining the correct ratcheting number.

Channel Creation with Contact Device (CCCD) Protocol. Consistent with Olvid's specifications, we model the protocol using seven rules, each corresponding to one step of the protocol, and similar how the protocol would be covered by different oracle cases in a game-based proof.

One interesting aspect of our model is how we model the complex preconditions and checks in Olvid's code. In Tamarin's modeling language, some preconditions for rules to be instantiated can be expressed by so-called restrictions, i.e., a rule can be *restricted* to only be enabled if a certain condition is met or some action previously happened. In Olvid's implementation, before party B starts executing the protocol, it deletes any existing instance of the same protocol between the same parties and devices, as well as any Oblivious Channel already established between its device and A's device. To accurately model this behavior, we define the restrictions *No_protocol_instance_exists* and *No_oblivious_channel_exists*:

```
restriction No_protocol_instance_exists:
  All a b deva devb #t.
    EnforceNoProtocolInstanceExists(a, b, deva, devb) @t
 ==>
    not(Ex #r pid. ProtocolInstance(pid, a, b, deva, devb) @r & #r<#t)
```

```
restriction No_oblivious_channel_exists:
  All a b deva devb #t.
    EnforceNoObliviousChannelExists(a, b, deva, devb) @t
 ==>
    not(Ex #r. CreateObliviousChannel(a, b, deva, devb) @r & #r<#t)
```

The first restriction models that when a rule containing the action *EnforceNoProtocolInstanceExists* is executed, the implementation checks that no instance of the same protocol between the same two parties and devices was running beforehand. The second restriction models that if a protocol step contains the action *EnforceNoObliviousChannelExists*, the implementation checks that no previous protocol execution reached the point where an oblivious channel was already generated by a party playing the role of A with B, using the same devices $devA$ and $devB$. These checks also apply when A performs the same $DELETE$ operations.

To model the $ABORT$ operation caused by the other party being inactive or not a contact, we introduce the following rule, which sets up the protocol under the assumption that both parties are active and already in each other's contact list. We then require the facts *ContactActive* and *ContactInContactList* as premises for all the protocol rules that include checks for inactivity or contact status. This guarantees that the protocol can only progress when these preconditions hold.

```
1 rule Setup_for_protocol:
2     []
3     -->
4     [ !ContactActive($actor), !ContactInContactList($actor, $peer) ]
```

To model the $ABORT$ behavior triggered by the reception of a previously received signature, we introduce the following restriction:

```
1 restriction uniqueness_signature:
2   All a b sigb #t.
3     EnforceNoSameSignatureReceived(a, b, sigb) @t
4  ==>
5     not(Ex #r. StoreSignature(a, b, sigb) @r & (#r < #t))
```

This ensures that whenever a protocol trace includes a rule that contains the action *EnforceNoSameSignatureReceived*, there exists no prior step in the execution trace in which the same signature was stored by the receiving party, similar to the checking condition in the implementation.

In the Olvid implementation, received KEM ciphertexts undergo validation, and malformed inputs result in a *DecryptionException* and subsequent abort. To reflect this behavior, we model decapsulation checks using explicit deconstructors, in combination with the embedded restriction on the predicate *ConditionCiphertext*:

```
1 ConditionCiphertext(ctx)
2 <==> Ex k pk r. ctx = encaps(k, pk, r)
```

This models that decapsulating arbitrary malformed data will cause an abort, but valid encapsulations of arbitrary keys with arbitrary randomness will not cause an abort.

Full Ratchet Protocol. Following Olvid's specifications, we model the protocol from Figure 7.5 using six rules: three for parties playing the role of

A and two for parties playing the role of B. In this setting, A's root key is the key to update. The execution of the protocol begins with two initialization rules, *FRP_Init_1* and *FRP_Init_2*. These rules ensure that the OBLIVIOUS CHANNEL is synchronized on both sides before any further steps are taken. No additional details about the FULL RATCHET PROTOCOL model are provided here, as its design largely follows the same modelling principles as the CCCD PROTOCOL, and any differences are not relevant outside the strict modelling context.

Adversarial capabilities. We perform several different analyses. As mentioned in the introduction, there is a strong difference between the technical security claims in [BF19], which are relatively weak, and informal claims in [Olv25a], which are extremely strong. To cover this, we both model the weaker technical claims about forward secrecy as well as the eCK security model, which is more representative of the strong informal claims. In the eCK setting, the adversary can corrupt long-term keys, such as KEM keys and signing keys, and reveal session keys (in our case, it can reveal session seeds and then retrieve the corresponding session keys). It can also reveal ephemeral KEM keys and the randomness values used in *KEM.Enc*. As the adversary is assumed to have full control over the network, it can intercept, modify, delete, and inject arbitrary messages. The model further allows the adversary to attempt Key Compromise Impersonation (KCI) attacks, whereby knowledge of a party's long-term key is exploited to impersonate other parties to that compromised party. Due to the fact that signatures in the implementation are malleable, in the CCCD PROTOCOL the adversary is also able to replace valid signatures, as well as send crafted *Ping* messages to trigger replay attacks. Additionally, in the FULL RATCHET PROTOCOL, we consider the adversary capable of compromising either the sender's seed or the receiver's seed used to ratchet the chain keys.

7.4.2 Positive findings

Authentication. For both the CCCD PROTOCOL and FULL RATCHET PROTOCOL, we model authentication between entities in TAMARIN as an *injective agreement*. This property captures that only the intended participants engage in the protocol within their assigned roles and that they share a consistent view of the agreed-upon terms.

In TAMARIN's syntax, such formal properties are specified as lemmas to be proven, and we show an example of the mutual injective agreement below. The final conjunct of the lemma is the injectivity condition, and it guarantees that each protocol instance of party A is uniquely matched to a single instance of party B. Informally, this injectivity corresponds to a uniqueness that models replay protection.

```
1  lemma mutual_authentication:
2    All p1 p2 dev1 dev2 pid2 k1 k2 #i.
3      Commit(pid2, p2, p1, dev2, dev1, k1, k2) @i &
4      Role(pid2, <p2, p1, dev2, dev1, 'Bob'>) @i &
5      not(Ex p #r. RevKemKeys(p) @r & #r < #i)
6      ==>
7      Ex pid1 #j.
8          Running(pid1, p1, p2, dev1, dev2, k1, k2) @j &
9          Role(pid1, <p1, p2, dev1, dev2, 'Alice'>) @j &
10         #j < #i
11     & not(Ex #t2.
12         Commit(pid2, p2, p1, dev2, dev1, k1, k2) @t2 &
13         not(#i=#t2))
```

The formalization above slightly simplifies the formal property in the artifact to provide more intuition.[4] The formula states that if an agent *p2*, executing in the role of party B, computes the keys *k1* and *k2*, and no adversary has, prior to this computation, obtained any long-term KEM private key of any agent nor any ephemeral KEM private key associated with any session, then there must exist an agent *p1*, in the role of party A, such that *p1* was already active prior to *p2* computing the keys.

Forward secrecy. In the Channel Creation with Contact Device Protocol and Full Ratchet Protocol, Olvid asserts that it provides perfect forward secrecy (FS). In Tamarin, we developed a series of secrecy lemmas, with varying levels of strength, to elucidate protocols' limitations related to FS. The results obtained from these lemmas are positive with respect to the standard notion of forward secrecy, in which the only patterns that lead to the disclosure of the session seed involve the disclosure of the long-term KEM keys of the participants, prior to the establishment of the seed by them. Additionally, we successfully verify a stronger definition of secrecy, shown below. Its strength is in considering the additional adversarial of revealing the seed of the considered session or of another session that has the same seed.

```
1  All actor peer devActor devPeer sidActor seed #i1 #i3.
2    SessionSeedEstablished(sidActor, actor, peer, devActor, devPeer, seed)
       @i1 &
3    Role(sidActor, <actor, peer, devActor, devPeer, 'Alice'>) @i1 &
4    K(seed) @i3 &
5    (All sid #i. RevRandomnessKem(sid) @ i  ==> F)
6    ==>
7    (Ex sid #i4. SessSeedRev(sid, seed) @i4) |
8    (Ex #i4. RevKemLtk(actor) @i4 & #i4 < #i1) |
9    (Ex #i4. RevKemLtk(peer) @i4 & #i4 < #i1)
```

However, we observe contrasting outcomes when considering security in stronger security models, as we will describe in subsection 7.4.3.

Post-compromise security. The Full Ratchet Protocol offers a set of security properties, one of which is post-compromise security. We model a CKA-like [ACD19] property by proving that if the adversary is passive with respect to a specific run of the full ratchet protocol (i.e., allowing the ratchet to heal), then even if the adversary can compromise long-term secrets but not the specific

[4]The actual property analyzed includes a more complex threat model which leads to more complicated "freshness conditions".

seed or ephemerals of the current run, then the new seed is secret. We formally model this security property in TAMARIN using the following lemma, which TAMARIN quickly successfully verifies, using a custom proof tactic to improve performance.

```
1 lemma PCS:
2   All actor peer devActor devPeer pidActor pidPeer seed #j #l #t.
3       UpdatedSendSeed(pidActor, actor, peer, devActor, devPeer, <seed,
            devActor>) @l &
4       UpdatedRecvSeed(pidPeer, peer, actor, devPeer, devActor, <seed,
            devActor>) @t &
5       K(seed) @j
6     ==>
7       (Ex #i. RevEphAny(pidActor)@i ) |
8       (Ex #i. RevEphAny(pidPeer)@i ) |
9       (Ex pidx #i. SessSeedRev(pidx, seed)@i ) |
10      (Ex a b #i. CompromiseRecvSeed(a, b, seed)@i ) |
11      (Ex a b #i. CompromiseSendSeed(a, b, seed)@i )
```

The verification confirms that the full ratchet protocol can effectively heal the ratcheted session.

However, there are two design aspects that weaken any potential PCS guarantees. First, as established experimentally already in [CFKN20], Olvid ultimately does not achieve PCS at the application layer, since sessions can be arbitrarily restarted based on long-term secrets. Second, unlike Signal's ratchet, Olvid does not "accumulate" entropy of all previous ratchets into the next full ratchet: the new seed replaces the old seed. This means that even if the session handling were to be improved, compromise of the ephemeral keys $(K1, K2)$ of the current session would always compromise the next seed, which would not be the case if Olvid would chain in the previous seed into a PRF-PRNG-like construction [ACD19] as Signal does.

Replay protection. In the CHANNEL CREATION WITH CONTACT DEVICE PROTOCOL, Olvid ensures replay protection by embedding a signature within the *Ping* message. Upon receiving this message, a participant checks whether the signature has been previously received. If it has, the protocol is aborted; otherwise, the signature is stored to prevent future replays. To verify the effectiveness of this strategy, we modeled a replay protection lemma in TAMARIN. Our analysis confirms that the approach works but also highlights certain drawbacks, which we discuss in section 7.5. Following Olvid's source code, we model malleable signatures.

The replay protection lemma specifies that once a participant receives and stores a signature associated with a specific pair of devices, any forged version of the same message's signature created by an adversary cannot be stored. In other words, only valid signatures are accepted and retained.

```
1 lemma replay_protection:
2  All pidActor actor peer devActor devPeer sig1 sig2 #t1 #t2 #t3.
3   Ping(pidActor, actor, peer, devActor, devPeer, sig1) @ t1 &
4   StoreSignature(peer, actor, sig1) @t2 &
5   ReplaceSignatureAndSendPing(actor, peer, devActor, devPeer, sig1, sig2)
        @t3
6   ==>
7   not(Ex #t4. StoreSignature(peer, actor, sig2) @t4)
```

KCI resistance. Key Compromise Impersonation (KCI) considers the scenario where an attacker compromises a party X's long-term key, and then exploits the protocol to impersonate as an arbitrary party towards X. The below property models this by checking for agreement while allowing compromise of the actor, which is reflected in the fact that the property should hold *unless* the long-term key of the peer is compromised. Agreement should thus even hold when the actor is compromised, ensure the absence of KCI attacks. We successfully verified this property with respect to an adversary that can reveal long-term keys but not short-term secrets.

```
1
2 lemma KCI_resistance:
3  All pidActor actor peer devActor devPeer k1 k2 #t1.
4   Commit(pidActor, actor, peer, devActor, devPeer, k1, k2) @t1 &
5   not(Ex #t. RevKemLtk(peer) @ t & #t < #t1)
6   ==>
7   Ex pidPeer #t2.
8     Running(pidPeer, peer, actor, devPeer, devActor, k1, k2) @t2
```

7.4.3 Negative findings

Session keys computed as output of authenticated key exchange are a function of four secret keys, namely the static keys and the ephemeral keys of each of the two parties. Strong security models like the extended Canetti-Krawczyk (eCK) model introduced in [LLM07] not only consider security of *past* sessions if some of those keys are compromised, but also mandate key secrecy if certain patterns of secret keys are compromised *during the attacked session*. This property is sometimes referred to as security under *maximal exposure (MEX)* attacks [FSXY12].

Ephemeral key reveal. One property mandated by eCK and MEX security is key secrecy with ephemeral keys of both parties revealed in the attacked session. This captures an attack scenario where the devices running the protocol employ a compromised or low-entropy randomness source. This scenario has been demonstrated to be relevant in the real world by, e.g., Debian's OpenSSL randomness disaster [Wei08]. We model this scenario in Tamarin and identify an attack trace. Specifically, we modeled an attack scenario where the compromise of both peers' ephemeral secret keys and randomness allows an adversary to compute the session seed (session keys).

Intuitively the reason for the existence of this attack is that both the CCCD Protocol and the Full Ratchet Protocol run ephemeral KEM-based key exchanges through an encrypted and authenticated channel, from which they inherit authentication. However, the session seed that is output from the protocol is independent of all static keys, so there is no way for those static keys to contribute to session-key confidentiality. One can consider this as the "dual of KCI", where instead of compromising the target's long-term keys, the adversary compromises *only* the target's randomness used for ephermeral secrets.

Static-ephemeral reveal. Another property mandated by eCK and MEX security is key secrecy with the static key of one party and the ephemeral key of the other party revealed in the attacked session. We model also this reveal pattern in TAMARIN and show that security indeed *does* hold because the lemma provided below is falsified, meaning that there is no trace that, thanks to this reveal pattern, allows the adversary to compute the session seed:

```
1
2 lemma static_ephemeral_reveal:
3 exists-trace
4 Ex I R pidI pidR devI devR seed #t1 #t2 #t3 #t4 #t5.
5   SessionSeedEstablished(pidI, I, R, devI, devR, seed) @ t1 &
6   SessionSeedEstablished(pidR, R, I, devR, devI, seed) @t2 &
7   RevKemLtk(I) @ t3 &
8   RevEKemSk(pidR, R, I, devR, devI) @ t4 &
9   not(Ex #t. RevEKemSk(pidI, I, R, devI, devR) @ t) &
10  not(Ex #t. RevKemLtk(R) @ t) &
11  not(Ex pid s #t. SessSeedRev(pid, s) @ t) &
12  not(Ex #t. RevRandomnessKem(pidI) @ t) &
13  K(seed) @ t5 & #t3 < #t5 & #t4 < #t5
```

An important aspect of this proof is that it relies on the design choice, present in Olvid, that ephemeral public keys are not accessible to the adversary. This assumption is justified in the protocol flow, since these keys are transmitted exclusively over the encrypted and authenticated ASYMMETRIC CHANNEL (in the CCCD PROTOCOL) or OBLIVIOUS CHANNEL (in both CCCD PROTOCOL and FULL RATCHET PROTOCOL). However, requiring public keys to be kept secret is an unusual assumption and may prove fragile when interfacing with implementation security. Implementors of cryptographic libraries typically assume that it is no problem to leak information about public keys through, e.g., timing information. We therefore also model the scenario where in addition to the static key of one party and the ephemeral key of the other party all public keys are revealed to the adversary.

```
1 lemma public_ephemeral_public_key_attack:
2 exists-trace
3   Ex I R pidI pidR devI devR seed #t1 #t2 #t3 #t4.
4     SessionSeedEstablished(pidI, I, R, devI, devR, seed) @t1 &
5     K(seed) @t2 &
6     not(Ex seed #i3. SessSeedRev(pidI, seed) @i3) &
7     not(Ex seed #i3. SessSeedRev(pidR, seed) @i3) &
8     not(Ex sid pa pb deva devb #t3 #t4. RevEKemSk(sid, pa, pb, deva, devb)
          @t3 & RevKemLtk(pa) @t4) &
9     not((Ex #t3 #t4. RevEKemSk(pidI, I, R, devI, devR) @t3 &
          RevEKemSk(pidR, R, I, devR, devI) @t4)) &
10    RevealRandomness(pidI, I, R, devI, devR) @t3 &
11    RevEKemPk(pidR, R, I, devR, devI) @t4
```

The four not-statements in the lemma ensure that only traces relevant to the specific attack are considered, filtering out those that do not contribute to the analysis. As expected, TAMARIN finds an attack trace for this scenario. Again, intuitively the reason for the attack is that session seeds are independent of static keys.

7.5 Source-code analysis and implementation security

In order to derive the formal specification of Olvid described in Section 7.4.1, we often relied on closely inspecting the Java source code of the Android client. In this context we also reviewed the code for security issues; we report the findings of this source-code analysis in this section. We emphasize that this analysis is limited to the Java source code targeting Android and does not cover the Swift implementation for iOS. Also, the analysis should not be confused with a complete and systematic source-code audit.

7.5.1 Positive findings

Code very close to specification. Overall, the code is clean and easy to read. In particular we appreciate the efforts that went into the very close matching between the source code and the specification [BF24]. Specifically, pseudocode in the specification and code in the actual implementation use the same function names and, where applicable, also the same variable names. This helped significantly with deriving our formal model of the cryptographic core of Olvid.

7.5.2 Negative findings

Non-Canonical Values. Both the Android and iOS versions of Olvid do not use established libraries for cryptographic primitives, but instead rely on their own implementations of all cryptographic primitives (written in Java and Swift, respectively). In the implementations of elliptic-curve-based primitives, we noticed several instances of assuming that values are in a (fully reduced) canonical representation without actually ensuring first that they are. Most notably, Olvid implements the Elliptic Curve Based Schnorr Digital Signature Algorithm (ECSDSA) [fSidI18], but omits range checks. This means that signatures in Olvid are malleable. As Olvid's replay protection relies on storing all signatures received in *Ping* messages at the beginning of the CCCD Protocol in a database, one might think that this malleability results in a straight-forward attack against replay protection. However, interestingly, we were not able to leverage this to an actual attack. What saves Olvid's replay-protection mechanism is once more the fact that all messages of the CCCD Protocol are sent through the Asymmetric Channel.

DoS vulnerability from replay protection. Storing signatures for replay protection however yields another attack vector, namely for a DoS attack that aims at filling up the database and thereby eventually exhausting the victim's memory and storage. This process is relatively slow. In an experiment with a Google Pixel 4a device as victim we were able to fill up the database at a rate of about 100 KB per minute from a single attacker device. We were able to roughly double this rate when adding a second attacker device. What makes this DOS

attack somewhat obnoxious is that it is persistent; restarting the application or device will not resolve the issue. Regularly emptying the database would clearly help, but also eliminate the replay protection. In Section 7.6 we briefly discuss a first mitigation, which consists of replacing signatures by hashes.

Timing leakage in scalar multiplication. Finally, and most critically, we found one instance of the open-source implementation of Olvid not following the constant-time programming policy, which is widely accepted as state-of-the-art for cryptographic implementations. More specifically, we find that the scalar multiplication implemented for EdwardCurve is not constant-time. Specifically, it doubles a point or adds a point to itself depending on the value of the scalar bit, which is evaluated by a conditional branch. Knowing each branch outcome leaks whether each bit is zero or one, allowing an attacker to reconstruct the scalar [YF14].

We present the vulnerable code snippet in Listing 7.1. The input `n`, which is the scalar, is evaluated by a conditional branch at line 4. The bodies of the branch are almost identical, except that the operands of some operations are different; we highlight differences in red. It follows a relaxed constant-time programming policy, which assumes a branch with similar bodies does not cause measurable timing differences [ACE+18].

```
1  scalarMultiplication(n, Y) {
2    ...
3  Loop on n :
4    if (n.testBit(i)) {
5      t3 = uR.add(wR).modPow(TWO, p);
6      t4 = uR.subtract(wR).modPow(TWO, p);
7      t5 = t3.subtract(t4).mod(p);
8      u2R = t3.multiply(t4).mod(p);
9      w2R = t5.multiply(t4.add(
10         c.multiply(t5))).mod(p);
11     uQ = uQplusR; wQ = wQplusR;
12     uR = u2R; wR = w2R;
13   } else {
14     t3 = uQ.add(wQ).modPow(TWO, p);
15     t4 = uQ.subtract(wQ).modPow(TWO, p);
16     t5 = t3.subtract(t4).mod(p);
17     u2Q = t3.multiply(t4).mod(p);
18     w2Q = t5.multiply(t4.add(
19         c.multiply(t5))).mod(p);
20     uQ = u2Q; wQ = w2Q;
21     uR = uQplusR; wR = wQplusR;
22   }
23 }
```

Listing 7.1: Vulnerable Scalar Multiplication

We show that this assumption does not hold here with a proof-of-concept attack on the vulnerable scalar multiplication. We first compile the exact same scalar multiplication taken from Olvid source code, which is written in Java, to native binary on Intel i7-10710U that runs Ubuntu 20.04.06LTS. Then we monitor two instruction locations, one for each branch body, with the classic `Flush+Reload` technique [YF14]. That is, the spy process keeps flushing the monitored instructions out of the cache and measuring the cost of reloading

them in a periodic manner. If the reload cost is small, it indicates that the victim process has accessed the flushed instruction which brings the instruction back the cache. Otherwise, the victim has not accessed the monitored instruction.

We pin the victim process to one core and the spy process to another core. We first execute the spy process, which keeps monitoring the two locations without knowing when the victim starts executing the scalar multiplication. Then, we execute the victim process which processes a random scalar n. Depending on the value of each scalar bit, the victim process executes one of the two branches and the spy process should observe the timing difference of reloading the two locations. If the scalar bit is one, the attacker will observe a small reload cost for instruction location in the throughput branch, and a large reload cost for instruction location under the else branch. Figure 7.6 presents partial of the results, where the x-axis is the scalar bit index and the y-axis is the reload cost. As shown in the figure, the timing difference between executing different branches is significant: if the victim executes the monitored instruction, the reload cost is always below 50 cycles, otherwise it is always above 100 cycles. According to the figure, the recovered bits are 1011110110.

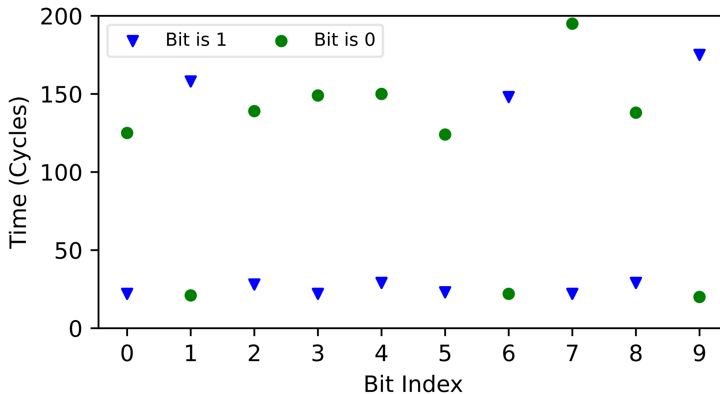


Figure 7.6: `Flush+Reload` attack on the scalar multiplication.

7.6 Conclusion and Future Work

Our formal analysis presented in Section 7.4 and our source-code and implementation-security analysis presented in Section 7.5 show that the cryptographic core of Olvid gets many things right, but still falls somewhat short of the goal of matching or even exceeding the state of the art in terms of security.

Some of the issues we found are fairly easy to address. This includes eliminating the timing leakage in elliptic-curve scalar multiplication (at least on source level; robust solutions to preserve this property through compilation for mainstream languages are still lacking [SLP+25, GPSar, GM25]). Also,

more careful canonicalization through modular reductions before range checks or other comparisons should be easy to include.

Other issues, most notably security under certain key-reveal patterns, will require updates to the Olvid protocol stack, which will break backwards compatibility. We recommend including the key output of the encapsulations to static keys in the session-key derivation in the CCCD Protocol. Similarly we recommend including the previous session keys in the derivation of fresh keys in the Full Ratchet Protocol. We are in a constructive dialogue with the Olvid developers that we hope will eventually result in a concrete proposal.

More broadly, there are further aspects of the Olvid ecosystem that we believe can be improved to increase security and trust, we reflect on those aspects in the following and also circle back to the use of (malleable) signatures for replay protection and the resulting DoS attack vector we discussed in Section 7.5.

Formal security specifications. The Olvid specification [BF24] only describe *how* Olvid functions, not *why* protocols are designed the way they are. For formal security analyses such as the one we present in this chapter but also for potential future work it would be very helpful, if the specification formally stated for each sub-protocol what assumptions is makes about inputs, employed subprotocols, and primitives, and what security goals it aims to achieve.

As one example, the use of the Full Ratchet Protocol clearly hints at post-compromise security (PCS) as a design goal of Olvid, but this is not stated or formalized anywhere. Also, as shown in [CFKN20], Olvid does not actually achieve PCS at application level. If PCS *is* a design goal, then changes will be required to extend the property from protocol to application level; if it is not a design goal, an obvious simplification of Olvid's cryptographic core would be to drop Full Ratchet Protocol altogether.

As another example, the CCCD Protocol at a first glance looks like an authenticated-key-exchange built from signatures and ephemeral KEM-based key exchange. However, the signatures do not actually contribute to authentication; instead, authentication is inherited from the encrypted and authenticated Asymmetric Channel through which all messages are sent. It is still not entirely clear to us if this is an intentional design decision or a "happy accident".

Unclear purpose of signatures. One use that signatures do have is to ensure replay protection during the Channel Creation with Contact Device Protocol. One would usually expect that this use case requires signatures to satisfy *strong* existential unforgeability under chosen message attacks (SUF-CMA). As explained in Section 7.5,this condition is not met in Olvid's current implementation; the replay-protection mechanism is saved by the fact that all messages of the CCCD Protocol are sent through the Asymmetric Channel. A more robust and efficient approach would be to replace signatures with second-preimage-resistant hashes. Using Tamarin, we analyzed the protocol's security properties after substituting signatures with such hashes and found

that all security properties we prove for Olvid with signatures remain intact, while replay protection is significantly strengthened, even in the absence of the Asymmetric Channel.

Additional benefits of using hashes instead of signatures would be improved computational performance and smaller storage requirements, as signatures have a size of 80 bytes, whereas hashes would occupy only 32 bytes. The latter would help as a first step towards mitigating the DOS attack vector described in Section 7.5. However, at the moment a signed ping message also triggers tearing down already existing channels [BF25]. When replacing signatures by hashes, this mechanism would need to be updated to use authentication provided by the Asymmetric Channel.

Anonymity claims too bold. Olvid's advertisement includes questionable statements about anonymity guarantees, for example *"Olvid is the only system that also encrypts metadata, thus guaranteeing the anonymity of interlocutors"*, or *"No third party could ever identify the participants, not even the server. No trace of any metadata."*. In fact, servers see a lot of metadata including recipient public keys (required to deliver messages), message sizes, and timing information of messages. The link between public keys and individuals is not immediately public, but will in many cases be rather trivial to establish, not only if server operators are at the same time also users of Olvid with their own contact list. Following common cryptographic terminology [PK01], it would certainly be more appropriate to speak about *pseudonymity* rather than anonymity and refrain from making statements about unavailability of metadata to the server.

Not completely open-source (yet). While the client code of Olvid is open source, the server code is currently not. Olvid lists four reasons why they did not yet make the server source code available[5]. First, they argue that because the server is untrusted, *"making the code open source would have no beneficial security impact"*. While we agree that having the client code available is much more important, we disagree that the server code is not relevant for security analysis. For example, it would be very useful to study the code to get a better understanding of what privacy or "anonymity" users can expect if a server is seized by law enforcement. The next two reasons are mostly relating to the fact that users are discouraged from running their own server until APIs have stabilized. This makes sense, but could also be achieved by clear warnings while still making the server code available for independent security analysis. The last reason is essentially that Olvid would like to have some security by obscurity against DOS attacks. Overall, we do not find any of the reasons fully convincing and encourage Olvid to release the server source code rather sooner than later.

[5]See `https://olvid.io/faq/server-and-open-source/`

7.6.1 Avenues for future work

One obvious avenue for future work is an analysis in the computational model. As mentioned before, in order to modularize this analysis, it would be very helpful to have formal security assumptions and goals defined for each sub-protocol in Olvid's specification.

Another direction for future work is to investigate security of Olvid beyond the cryptographic core, including, e.g., the details of group messaging and session handling.

As mentioned in Section 7.5, our source-code analysis is not a full systematic code audit. Also, it is limited to the Java code targeting Android and does not cover the Swift implementation targeting iOS. A full code audit of all Olvid clients would certainly be useful to improve the implementation and increase trust, but is beyond the scope of academic research.

Finally, while we informally criticize the anonymity claims as too bold, we leave a formalization of what privacy properties Olvid *does* achieve to future work.

Chapter 8

Conclusion and Outlook

Within this thesis we implemented post-quantum solutions for resource-constrained environments. For all investigated cases it was in fact possible to employ PQC, but it was necessary to take the system's exact characteristics into account. Both implementation and algorithm had to be tweaked to the use case, giving valuable, generalizable insights. The exact cost of our post-quantum solutions was detailed.

We would like to conclude with a brief outlook on further research directions the author deems relevant. Work on this thesis began when it was still unclear which algorithms would be selected by NIST. With the first standards in place, and further algorithms scheduled for standardization, the field of post-quantum cryptography has reached a milestone. However, it seems that there is still much to do.

More Implementation and Algorithmic Tweaks. It appears that lattice-based algorithms, such as Kyber and Dilithium, will play a dominant role for common scenarios. In fact, large institutions such as Google [Han24], Cloudflare [Wes23], Signal [Kre24] or Apple [Eng24] have already included them into products. Unfortunately, in the embedded world, few scenarios are "common". Custom solutions will have to be found in order to encompass PQC into systems that often times already struggle with the much cheaper ECC. These solutions could include tweaked algorithms (such as presented in Chapter 6). Much attention in the literature has been given to aligning a certain algorithm to a given platform using implementation tricks. It is unclear if this will suffice for all circumstances. Designing algorithms that are flexible enough so that they can be tweaked not only on an implementation level, but on an algorithmic level, seems an interesting avenue of research. This has to be done with caution of course, as it can be very dangerous to tweak fragile algorithms. Of all NIST-selected algorithms, only SPHINCS+ comes with this maximum of flexibility.

Embedded Authentication. Embedded systems tend to have much longer development and life cycles than software products. While it is understandable,

that due to *harvest now, decrypt later* threats, key establishment mechanisms found adopters early on, authentication also plays a vital role. In the embedded realm, the need to upgrade to quantum-safe authentication is more pressing than it seems. Software products might be able to update trivially once quantum computers become a reality. However, long-lasting embedded systems are developed and deployed constantly. Some with decade-long expected lifetimes. My government-issued passport, for example, has a validity of ten years and uses ECC for authentication. More research into which embedded scenarios require post-quantum solutions and how those solutions could look like is needed. The ongoing NIST selection process for additional post-quantum signatures would benefit from this research as well.

Formal Verification. Cryptography is complex. That has not changed for the better with the advent of post-quantum cryptography. Compared to decades of trial-and-error learning on how to improve pre-quantum cryptography implementation, post-quantum is still in its infancy. How this can affect implementation security was recently put on display via a side-channel attack on Kyber's reference implementation [Fir]. The side channel was introduced through counter-intuitive, but legal, compiler behavior. Formally verifying implementations and therefore assuring correct algorithms and implementations is a long-awaited dream of computer scientists. There are promising approaches (see e.g. Chapter 4) to this. However, as they add complexity to an already complex subject, it is not always easy to get them right or understand what exact guarantees are given. Research on how to enhance usability of fomal verification methods is therefore just as important as developing those methods.

Post-Quantum Education. Trust is very difficult to gain, but extremely easy to lose. In countless conversations with engineers from industry I have heard reservations against the use of PQC. News about broken schemes like Rainbow [Beu22] or SIKE [CD22] spread far beyond the usual circle of cryptographers. Combined with the abstract threat of a quantum computer that few people are able to imagine, this makes developers doubtful. Especially in the usually-technology-conservative embedded world, where additional hardware upgrades are expensive, this doubt could lead to fewer adoption. Government-mandated standards aren't necessarily a solution in a competitive industry operating internationally. At least not, if engineers and developers think of PQC as expensive and as adding more security issues than it counters. Making PQC more transparent and approachable through means of education is therefore of great importance for the field. Research into how this transparency could best be achieved is needed.

Acronyms

The list below presents commonly used acronyms in alphabetical order as referenced throughout this thesis. Names of schemes or algorithms are excluded.

AEAD Authenticated Encryption with Associated Data

AKE Authenticated Key Exchange

AWS Amazon Web Services

CPU Central Processing Unit

CTAP Client to Authenticator Protocol

DH Diffie-Hellman

DRAM Dynamic RAM

ECC Elliptic Curve Cryptography

eOS Embedded Operating System

FIDO Fast IDentity Online

FPU Floating Point Unit

FRP Full Ratchet Protocol

HKDF Hash-based Key Derivation Function

ISA instruction set architectures

ISR interrupt service routine

KDF Key Derviation Function

KEM Key-Encapsulation Mechanism

LWE Learning With Errors

MAC Message authentication codes

MCU microcontroller unit

MLWE Module Learning With Errors

MSIS Module Shortest Integer Solution

ORAM Oblivious RAM

OT Oblivious Transfer

PKE Public Key Encryption

PQC Post-Quantum Crypto

PSK Pre Shared Key

RAM Random-Access Memory

ROM Read-Only Memory

SIMD Single-Instruction Multiple-Data

SRAM Static RAM

TLS Transport Layer Security

TPM Trusted Platform Module

XOF EXtendable Output Function

Bibliography

[3rd15] 3rd Generation Partnership Project (3GPP). The mobile broadband standard specification release 13. Technical report, 3GPP, September 2015. `https://www.3gpp.org/ftp/Information/WORK_PLAN/Description_Releases/Rel-13_description_20150917.zip` (accessed 2022-02-04).

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1807–1823. ACM Press, October / November 2017.

[ABB+20a] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy*, pages 965–982. IEEE Computer Society Press, May 2020.

[ABB+20b] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS+. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. available at `https://sphincs.org/`.

[ABB+22] J Aumasson, D Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, et al. SPHINCS+-submission to the NIST post-quantum project, v3. 1. *NIST PQC round*, 3, 2022.

[ABC+21] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. An extended account of trace-relating compiler correctness and secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(4):14:1–14:48, 2021.

[ABCP24] Martin R. Albrecht, Matilda Backendal, Daniele Coppola, and Kenneth G. Paterson. Share with care: Breaking E2EE in nextcloud. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, pages 828–840. IEEE, 2024. https://eprint.iacr.org/2024/546.

[Abd20] Michel Abdalla. Security Analysis of Olvid's SAS-based Trust Establishment Protocol. Cryptology ePrint Archive, Paper 2020/808, 2020. https://eprint.iacr.org/2020/808.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, volume 11476 of *LNCS*, pages 129–158. Springer, 2019. https://eprint.iacr.org/2018/1037.

[ACDJ23] Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable cryptographic vulnerabilities in matrix. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 164–181. IEEE, 2023. https://doi.org/10.1109/SP46215.2023.10351027.

[ACE+18] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. Sidetrail: Verifying time-balancing of cryptosystems. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments*, volume 11294 of *LNCS*, pages 215–228. Springer, 2018. https://d1.awsstatic.com/Security/pdfs/SideTrail_Verifying_Time_Balancing_of_Cryptosystems.pdf.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, volume 13508 of *LNCS*, pages 34–68. Springer, 2022. https://eprint.iacr.org/2020/1327.pdf.

[AMPS22] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. In

2022 IEEE Symposium on Security and Privacy (SP), pages 87–106. IEEE, 2022. https://doi.org/10.1109/SP46214.2022.9833666.

[ANS20] ANSSI. Rapport de certification anssi-cspn-2020/30: Olvid version 0.8.2 pour ios, 2020. https://cyber.gouv.fr/sites/default/files/2020/10/anssi-cspn-2020_30fr.pdf.

[ANS24] ANSSI. ANSSI views on the post-quantum cryptography transition, 2024. ttps://cyber.gouv.fr/sites/default/files/document/follow_up_position_paper_on_post_quantum_cryptography.pdf (accessed 2025-01-24).

[Aut] Libsodium Authors. Ed25519 implementation. https://github.com/jedisct1/libsodium/blob/59a98bc7f9d507175f551a53bfc0b2081f06e3ba/src/libsodium/include/sodium/crypto_sign_ed25519.h#L34. (accessed 2025-01-29).

[Bar23a] Cesar Eduardo Barros. Clear On Drop Source Code, 2023. https://github.com/cesarb/clear_on_drop (accessed 2023-07-15).

[Bar23b] Jean-No el Barrot. Tweet About Olvid, 2023. https://x.com/jnbarrot/status/1729964589742219712 (accessed 2025-03-07).

[BBB+24] Daniel J Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. *Cryptology ePrint Archive*, 2024. https://eprint.iacr.org/2024/1049.

[BBL+23] Olivier Blazy, Ioana Boureanu, Pascal Lafourcade, Cristina Onete, and Léo Robert. How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5917–5934. USENIX Association, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/blazy.

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570. IEEE Computer Society Press, May 2015.

[BDF+15] Thomas Baignères, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrède Lepoint, and Matthieu Rivain. Trap me if you can – million dollar curve. Cryptology ePrint Archive, Report 2015/1249, 2015. https://eprint.iacr.org/2015/1249.

[BDG+14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A Tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer International Publishing, 2014.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. Xmss-a practical forward secure signature scheme based on minimal security assumptions. In *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29–December 2, 2011. Proceedings 4*, pages 117–129. Springer, 2011.

[BDH+18] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1383–1396. ACM, 2018. https://hal.science/hal-01898050.

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer-Verlag Berlin Heidelberg, 2011. https://cryptojedi.org/papers/#ed25519.

[Ber06a] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[Ber06b] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer-Verlag Berlin Heidelberg, 2006. http://cr.yp.to/papers.html#curve25519.

[Ber09] Daniel J. Bernstein. *Introduction to post-quantum cryptography*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Ber22] Daniel J Bernstein. Multi-ciphertext security degradation for lattices. *Cryptology ePrint Archive*, 2022. https://eprint.iacr.org/2022/1580.

[Ber23] Daniel J Bernstein. Asymptotics of hybrid primal lattice attacks. *Cryptology ePrint Archive*, 2023. https://eprint.iacr.org/2023/1892.

[Beu22] Ward Beullens. Breaking rainbow takes a weekend on a laptop. In *Annual International Cryptology Conference*, pages 464–479. Springer, 2022.

[BF19] Thomas Baignères and Matthieu Finiasz. Security models of mobile messaging apps and applications of cryptography in communications, 2019. https://olvid.io/assets/documents/2019-06-12_Olvid_GDR-Securite-Informatique.pdf (accessed 2025-03-11).

[BF24] Thomas Baignères and Matthieu Finiasz. Specifications of Olvid – application and server, 2024. https://olvid.io/assets/documents/2024-10-07_Olvid-specifications.pdf (accessed 2024-12-20).

[BF25] Thomas Baignères and Matthieu Finiasz. Personal communication, 2025.

[BGHZ11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Heidelberg, August 2011.

[BGLP21] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Structured leakage and applications to cryptographic constant-time and cost. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 462–476. ACM Press, November 2021.

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical Stateless Hash-Based Signatures. volume 9056 of *LNCS*, pages 368–397. Springer, 2015.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ Signature Framework. In *ACM CCS*. ACM, 2019.

[BJKS24] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. Formal verification of the pqxdh post-quantum key agreement protocol for end-to-end secure messaging. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 469–486. USENIX Association, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/bhargavan.

[BK25] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST SP 800-90A, 2025. https://csrc.nist.gov/pubs/sp/800/90/a/r1/final (accessed 2025-08-20).

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *Progress in Cryptology – Africacrypt 2019*, LNCS, pages 209–228. Springer, 2019. https://eprint.iacr.org/2019/489.

[BLD+20] Shi Bai, Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. available at https://pq-crystals.org/dilithium/.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, 2017. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[BPSV19] Ward Beullens, Bart Preneel, Alan Szepieniec, and Frederik Vercauteren. LUOV. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[BR93] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO' 93*, volume 773 of *LNCS*, pages 129–158. Springer, 1993. https://cseweb.ucsd.edu/~mihir/papers/eakd.pdf.

[Bri23] Brian Smith. ring Source Code, 2023. https://github.com/briansmith/ring/commit/

b76f52c03a667c2ac6793ff566b55ad060421759 (accessed 2023-07-15).

[BRS22] Joppe W Bos, Joost Renes, and Amber Sprenkels. Dilithium for memory constrained devices. In *International Conference on Cryptology in Africa*, pages 217–235. Springer, 2022.

[BSKNS20] Kevin Bürstinghaus-Steinbach, Christoph Krauß, Ruben Niederhagen, and Michael Schneider. Post-quantum TLS on embedded systems: Integrating and evaluating kyber and SPHINCS+ with mbed TLS. In Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese, editors, *ASIACCS 20*, pages 841–852. ACM Press, October 2020.

[BSTP21] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV standard: Break, fix, verify. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1766–1781. IEEE, 2021. https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00037.

[BvdPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. “Ooh aah... just a little bit”: A small amount of side channel can go a long way. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014. https://link.springer.com/content/pdf/10.1007/978-3-662-44709-3_5.pdf.

[CC21] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):125–148, 2021. https://eprint.iacr.org/2021/492.

[CD22] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Report 2022/975, 2022. https://eprint.iacr.org/2022/975.

[CDJZ23] Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. Automated analysis of protocols that use authenticated encryption: How subtle aead differences can impact protocol security. In *USENIX Security Symposium*, pages 5935–5952. USENIX Association, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-protocols.

[CFKN20] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone detection in secure messaging: Improving post-compromise security in practice. In *Pro-*

ceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 1481–1495. ACM, 2020. `https://people.cispa.io/cas.cremers/downloads/papers/CFKN2020-messaging_cloning.pdf`.

[CFM+20] A. Casanova, J.-C. Faugère, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem. GeMSS. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. available at `https://www-polsys.lip6.fr/Links/NIST/GeMSS.html`.

[cfs] Cloudflare - manage active sessions. `https://developers.cloudflare.com/fundamentals/setup/account/account-security/manage-active-sessions/` (accessed 2025-01-29).

[CFS+21] Sofía Celi, Armando Faz-Hernández, Nick Sullivan, Goutam Tamvada, Luke Valenta, Thom Wiggers, Bas Westerbaan, and Christopher A. Wood. Implementing and measuring KEMTLS. In Patrick Longa and Carla Ràfols, editors, *LATINCRYPT 2021*, volume 12912 of *LNCS*, pages 88–107. Springer, Heidelberg, October 2021.

[CGCD+17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017. `https://doi.org/10.1109/EuroSP.2017.27`, extended version at `https://eprint.iacr.org/2016/1013.pdf`.

[CGCG16] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. Post-Compromise Security. Cryptology ePrint Archive, Paper 2016/221, 2016. `https://eprint.iacr.org/2016/221`.

[CGL06] Denis Charles, Eyal Goren, and Kristin Lauter. Cryptographic hash functions from expander graphs. Cryptology ePrint Archive, Paper 2006/021, 2006. `https://eprint.iacr.org/2006/021`.

[Cha17] Roderick Chapman. Sanitizing sensitive data: How to get it right (or at least less wrong…). In Johann Blieberger and Markus Bader, editors, *Reliable Software Technologies – Ada-Europe 2017*, volume 10300 of *Lecture Notes in Computer Science*, pages 37–52. Springer International Publishing, 2017.

[Che24] Yilei Chen. Quantum algorithms for lattice problems. *Cryptology ePrint Archive*, 2024. `https://eprint.iacr.org/2024/555`.

[CHH+17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788. ACM, 2017. https://doi.org/10.1145/3133956.313406.

[CHSvdM16] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485. IEEE, 2016. https://doi.org/10.1109/SP.2016.35.

[CHSW22] Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. A tale of two models: Formal verification of KEMTLS via tamarin. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, volume 13556 of *LNCS*, pages 63–83. Springer, 2022. https://eprint.iacr.org/2022/1111.

[CJN22] Cas Cremers, Charlie Jacomme, and Aurora Naska. Formal analysis of session-handling in secure messaging: Lifting security from sessions to conversations. Cryptology ePrint Archive, Paper 2022/1710, 2022. https://eprint.iacr.org/2022/1710.

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. Cryptology ePrint Archive, Paper 2001/040, 2001. https://eprint.iacr.org/2001/040.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action. In *Advances in Cryptology – ASIACRYPT 2018*, LNCS, pages 395–427. Springer, 2018. https://eprint.iacr.org/2018/383.

[CM05] S. Chong and A.C. Myers. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 241–254, 2005. https://people.seas.harvard.edu/~chong/pubs/csfw05_erasure.pdf.

[CMN24] Cas Cremers, Niklas Medinger, and Aurora Naska. Impossibility results for post-compromise security in real-world communication systems. Cryptology ePrint Archive, Paper 2024/1886, 2024. https://eprint.iacr.org/2024/1886.

[Con21] Dilithium Contributors. CRYSTALS-Dilithium specification v3.1. https://pq-crystals.org/dilithium/data/dilithium-specification.pdf, 2021. (accessed 2025-02-03).

[Con22] Connectivity Standards Alliance. Build with Matter, 2022. https://buildwithmatter.com (accessed 2022-05-16).

[Cou06] Jean-Marc Couveignes. Hard homogeneous spaces. Cryptology ePrint Archive, Report 2006/291, 2006. https://eprint.iacr.org/2006/291.

[CPGR05] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In Patrick D. McDaniel, editor, *USENIX Security 2005*. USENIX Association, July / August 2005.

[CSJ+19] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a DSL for timing-sensitive computation. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 174–189. ACM, 2019.

[CVE14] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., 2014. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (accessed 2023-07-15).

[Dal23] Dalek Authors. Dalek x25519 Source Code, 2023. https://github.com/dalek-cryptography/x25519-dalek/blob/2.0.0-rc.3/src/x25519.rs#L73 (accessed 2023-07-15).

[DBR22] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Transactions on Privacy and Security*, 26(2), 2022. https://binsec.github.io/assets/publications/papers/2022-tops.pdf.

[DCK+19] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2019. https://www.pqcrainbow.org/.

[DCK+20] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. available at https://www.pqcrainbow.org/.

[DDVY21] Jintai Ding, Joshua Deaton, Vishakha, and Bo-Yin Yang. The nested subset differential attack: a practical direct attack against luov which forges a signature within 210 minutes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 329–347. Springer, 2021.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34:1–42, 2021.

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

[Don17] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *NDSS 2017*. The Internet Society, February / March 2017.

[DPS15] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. The correctness-security gap in compiler optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pages 73–87. IEEE Computer Society, 2015.

[DvW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[EGHP09] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. Microeliece: Mceliece for embedded devices. In Christophe Clavier and Kris Gaj, editors, *CHES*, pages 49–64. Springer, 2009.

[Eng24] Apple Security Engineering. iMessage with PQ3: The new state of the art in quantum-secure messaging at scale. `https://security.apple.com/blog/imessage-pq3/`, 2024. (accessed 2025-02-03).

[FG24] Rune Fiedler and Felix Günther. Security analysis of signal's PQXDH handshake. Cryptology ePrint Archive, Paper 2024/702, 2024. `https://eprint.iacr.org/2024/702`.

[FHK+18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. Falcon: Fast-fourier lattice-based compact signatures over NTRU. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5):1–75, 2018.

[FIDa] FIDO Alliance. Authentication Specifications. https://fidoalliance.org/specifications/download/. (accessed 2025-02-03).

[FIDb] FIDO Alliance. CTAP2 specification. https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html. (accessed 2025-01-29).

[FIDc] FIDO Alliance. FIDO security reference. https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-rd-20210525.html. (accessed 2025-01-29).

[Fir] Cath Firmin. PQShield plugs timing leaks in Kyber / ML-KEM to improve PQC implementation maturity — PQShield — pqshield.com. https://pqshield.com/pqshield-plugs-timing-leaks-in-kyber/ (accessed 2025-02-03).

[fIS24] Federal Office for Information Security. Kryptografie quantensicher gestalten - quantum secure cryptography (german). https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Broschueren/Kryptografie-quantensicher-gestalten.pdf, 2024. (accessed 2025-02-17).

[FJ24] Rune Fiedler and Christian Janson. A deniability analysis of signal's initial handshake PQXDH. Cryptology ePrint Archive, Paper 2024/741, 2024. https://eprint.iacr.org/2024/741.

[FKL+20] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In *Advances in Cryptology – ASIACRYPT 2020*, LNCS, pages 64–93. Springer, 2020. https://eprint.iacr.org/2020/1240.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology – CRYPTO 1999*, LNCS, pages 537–554. Springer, 1999. http://dx.doi.org/10.1007/3-540-48405-1_34.

[For23] Formosa Crypto Team. Libjade, 2023. https://github.com/formosa-crypto/libjade (accessed 2023-07-15).

[fSidI18] Bundesamt fur Sicherheit in der Informationstechnik. Elliptic curve cryptography, 2018. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_V-2-1_pdf.pdf (accessed 2025-04-13).

[FSXY12] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Strongly secure authenticated key exchange from factoring, codes, and lattices. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *Public-Key Cryptography – PKC 2012*, LNCS, pages 467–484. Springer, 2012. https://eprint.iacr.org/2012/211.pdf.

[FWC16] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1505–1515. ACM, 2016. https://dl.acm.org/doi/pdf/10.1145/2976749.2978400.

[GCC23] GCC Authors. The GCC Documentation – Static Stack Usage Analysis, 2023. https://gcc.gnu.org/onlinedocs/gnat_ugn/Static-Stack-Usage-Analysis.html (accessed 2023-07-13).

[GHK+21] Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. Verifying Post-Quantum signatures in 8 kB of RAM. In *Post-Quantum Cryptography – PQCrypto 2021*, LNCS, pages 215–233. Springer, 2021. https://eprint.iacr.org/2021/662.

[GKP+23] Diana Ghinea, Fabian Kaczmarczyck, Jennifer Pullman, Julien Cretin, Stefan Kölbl, Rafael Misoczki, Jean-Michel Picod, Luca Invernizzi, and Elie Bursztein. Hybrid post-quantum signatures in hardware security keys. In *International Conference on Applied Cryptography and Network Security*, pages 480–499. Springer, 2023.

[GKPM18] Aymeric Genêt, Matthias J Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on SPHINCS. *Cryptology ePrint Archive*, 2018. https://eprint.iacr.org/2018/674.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2020. https://eprint.iacr.org/2020/1278.

[GLF+21] Tasopoulos George, Jinhui Li, Apostolos P. Fournaris, Raymond K. Zhao, Amin Sakzad, and Ron Steinfeld. Performance evaluation of post-quantum TLS 1.3 on embedded systems. Cryptology ePrint Archive, Report 2021/1553, 2021. https://eprint.iacr.org/2021/1553.

[GM25] Antoine Geimer and Clémentine Maurice. Fun with flags: How compilers break and fix constant-time code. arXiv report 2507.06112, 2025. https://arxiv.org/abs/2507.06112.

[Gnu23] GnuTLS Authors. GnuTLS Source Code, 2023. https://github.com/gnutls/gnutls/blob/gnutls_3_6_12/lib/safe-memfuncs.c (accessed 2023-07-15).

[Gon25] Ruben Gonzalez. Stateless hash-based signatures for post-quantum security keys. In *International Conference on Applied Cryptography and Network Security*, volume 15654 of *LNCS*. Springer, 2025. https://eprint.iacr.org/2025/298.

[Goo] Google. Session length for google services. https://support.google.com/a/answer/7576830 (accessed 2025-01-29).

[GPSar] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. Do compilers break constant-time guarantees? In Pedro Moreno-Sánchez Christina Garman, editor, *Financial Cryptography and Data Security*, LNCS. Springer, to appear. https://fc25.ifca.ai/preproceedings/13.pdf.

[GPTY18] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *Applied Cryptography and Network Security*, volume 10892 of *LNCS*, pages 83–102. Springer, 2018. https://eprint.iacr.org/2017/806.pdf.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *ACM STOC*, STOC '08, page 197–206. ACM, 2008.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC 1996*, pages 212–219, 1996. https://arxiv.org/abs/quant-ph/9605043.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912. USENIX Association, 2015. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[Gut16] Daniel Gutson. Proposal: Zero the local stack on function exit, 2016. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=69976 (accessed 2023-07-23).

[GVY17] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several

real-world applications of Curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 845–858. ACM, 2017. `https://eprint.iacr.org/2017/806.pdf`.

[GW22] Ruben Gonzalez and Thom Wiggers. KEMTLS vs. post-quantum TLS: Performance on embedded systems. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 99–117. Springer, 2022. `https://eprint.iacr.org/2022/1712`.

[Han24] Royal Hansen. Post-Quantum Cryptography: Standards and Progress — security.googleblog.com, 2024. `https://security.googleblog.com/2024/08/post-quantum-cryptography-standards.html` (accessed 2025-02-03).

[HBD+22] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`.

[HBG+18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018. `https://rfc-editor.org/rfc/rfc8391.txt`.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[HK22] Andreas Hülsing and Mikhail Kudinov. Recovering the tight security proof of SPHINCS+. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–33. Springer, 2022.

[HLP11] Stephen Hemminger, Fabio Ludovici, and Hagen Paul Pfeiffer, November 2011. `https://man7.org/linux/man-pages/man8/tc-netem.8.html` (accessed 2025-08-02).

[Hop19] Andrew Hopkins. Post-quantum TLS now supported in AWS KMS. Amazon AWS Security Blog, November 2019. `https://aws.amazon.com/blogs/security/post-`

quantum-tls-now-supported-in-aws-kms/ (accessed 2022-05-20).

[HRB13] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for xmss mt. In *Security Engineering and Intelligence Informatics: CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings 8*, pages 194–208. Springer, 2013.

[HRS+09] Olaf Henniger, Alastair Ruddle, Hervé Seudié, Benjamin Weyl, M. Wolf, and T. Wollinger. Securing vehicular on-board IT systems: The EVITA project. In *25th Joint VDI/VW Automotive Security Conference*, October 2009.

[HRS16] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS. In *PKC*, page 446–470. Springer, 2016.

[HS08] Sebastian Hunt and David Sands. Just forget it – the semantics and enforcement of information erasure. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2008.

[Hül13] Andreas Hülsing. W-ots+–shorter signatures for hash-based signature schemes. In *Progress in Cryptology–AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings 6*, pages 173–188. Springer, 2013.

[IAIES14] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *Research in Attacks, Intrusions and Defenses – RAID 2014*, volume 8688 of *LNCS*, pages 299–319. Springer, 2014. https://eprint.iacr.org/2014/435.pdf.

[Int17] International Organization for Standardization. The C17 Programming Language Standard, 2017. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf (accessed 2025-08-02).

[JCCGS19] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2165–2180. ACM, 2019. https://eprint.iacr.org/2019/779.

[JY02] Marc Joye and Sung-Ming Yen. The Montgomery powering ladder. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *LNCS*, pages 291–302. Springer, 2002. https://link.springer.com/content/pdf/10.1007/3-540-36400-5_22.pdf.

[KGB+18] Matthias J Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. Differential power analysis of XMSS and SPHINCS. In *Constructive Side-Channel Analysis and Secure Design: 9th International Workshop, COSADE 2018, Singapore, April 23–24, 2018, Proceedings 9*, pages 168–188. Springer, 2018.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE Computer Society Press, May 2019.

[KJPL24] Jaehyeon Kwak, Yunseong Jang, Jeewon Park, and Hanho Lee. SLH-DSA-based digital signature and verification FPGA system. *Transactions on Semiconductor Engineering*, 2(4):69–77, 2024.

[KLS+19] Krzyszstof Kwiatkowski, Adam Langley, Nick Sullivan, Dave Levin, Alan Mislove, and Luke Valenta. Measuring TLS key exchange with post-quantum KEM, August 2019. https://csrc.nist.gov/Presentations/2019/measuring-tls-key-exchange-with-post-quantum-kem.

[KP22] Stefan Kölbl and Jade Philipoom. A note on SPHINCS+ parameter sets. *Cryptology ePrint Archive*, 2022. https://eprint.iacr.org/2022/1725.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. volume 1592 of *LNCS*, pages 206–222. Springer, 1999.

[Kra10a] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, August 2010.

[Kra10b] Hugo Krawczyk. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, RFC Editor, May 2010.

[Kre24] Ehren Kret. Quantum Resistance and the Signal Protocol — signal.org. https://signal.org/blog/pqxdh/, 2024. (accessed 2025-02-03).

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KRSS19] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Workshop Record of the Second NIST PQC Standardization Conference, 2019.

[KSS24] Patrick Karl, Jonas Schupp, and Georg Sigl. The impact of hash primitives and communication overhead for hardware-accelerated SPHINCS+. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 221–239. Springer, 2024.

[Kuh18] Wouter Kuhnen. OPTLS revisited. Master's thesis, Radboud University, 2018. https://www.ru.nl/publish/pages/769526/thesis-final.pdf.

[KW16] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 81–96, 2016.

[KZ20] Daniel Kales and Greg Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. In *International Conference on Cryptology and Network Security*, pages 3–22. Springer, 2020.

[Lam79] Leslie Lamport. Constructing digital signatures from a one way function. 1979.

[Lan18] Adam Langley. CECPQ2. ImperialViolet, December 2018. https://www.imperialviolet.org/2018/12/12/cecpq2.html (accessed 2021-02-16).

[Lan19] Adam Langley. Real-world measurements of structured-lattices and supersingular isogenies in TLS. ImperialViolet, October 2019. https://www.imperialviolet.org/2019/10/30/pqsivssl.html (accessed 2021-02-16).

[Lan25] Tanja Lange. Hash-based signatures. In *Encyclopedia of Cryptography, Security and Privacy*, pages 1110–1112. Springer, 2025.

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and

Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[Lib23] Libsodium Authors. Libsodium Source Code, 2023. https://github.com/jedisct1/libsodium/blob/master/src/libsodium/sodium/utils.c (accessed 2023-07-15).

[LLM07] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Provable Security*, volume 4784 of *LNCS*, pages 1–16. Springer, 2007. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/strongake-submitted.pdf.

[LSB24] Felix Linker, Ralf Sasse, and David Basin. A formal analysis of apple's iMessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/1395, 2024. https://eprint.iacr.org/2024/1395.

[LWL+24] Yan Lin, Joshua Wong, Xiang Li, Haoyu Ma, and Debin Gao. Peep with a mirror: Breaking the integrity of Android app sandboxing via unprivileged cache side channel. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2119–2135. USENIX Association, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/lin-yan.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT*, pages 598–616. Springer, 2009.

[LZJZ22] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6):122:1–122:37, 2022. https://arxiv.org/pdf/2103.14244.

[MAB+18] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. Hamming quasi-cyclic (hqc). 2(4):13, 2018. available at https://pqc-hqc.org/.

[mbe] mbed TLS. https://www.trustedfirmware.org/projects/mbed-tls/ (accessed 2022-04-29).

[McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. The deep space network progress report 42-44, Jet Propulsion Laboratory, California Institute of Technol-

ogy, January/February 1978. https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, April 2019. https://rfc-editor.org/rfc/rfc8554.txt.

[Mer89] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[Mer90] Ralph Merkle. A certified digital signature. volume 435 of *LNCS*, pages 218–238. Springer, 1990.

[MF21] Arno Mittelbach and Marc Fischlin. The theory of hash functions and random oracles. *An Approach to Modern Cryptography, Cham: Springer Nature*, 2021.

[MI88] Tsutomu Matsumoto and Hideki Imai. Public quadratic polynominal-tuples for efficient signature-verification and message-encryption. volume 330 of *Lecture Notes in Computer Science*, pages 419–453. Springer, 1988.

[Mic] Microsoft. Session timeouts for ms 365. https://learn.microsoft.com/en-us/microsoft-365/enterprise/session-timeouts (accessed 2025-01-29).

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–243, 1987. https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf.

[MRAP23] Lenka Mareková Martin R. Albrecht, Miro Haller and Kenneth G. Paterson. Caveat implementor! key recovery attacks on MEGA. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, volume 14008 of *LNCS*, pages 190–218. Springer, 2023. https://mega-caveat.github.io/.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013. https://people.cispa.io/cas.cremers/downloads/papers/MSCB2013-Tamarin.pdf.

[Nat13] National Institute of Standards and Technology. FIPS186-4: Digital Signature Standard (DSS), 2013. https://doi.org/10.6028/NIST.FIPS.186-4.

[Nat16] National Institute for Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf (accessed 2025-08-02).

[Nat18] National Institute of Standards and Technology. NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography, 2018. https://doi.org/10.6028/NIST.SP.800-56Ar3.

[Nat19] National Institute of Standards and Technology. NIST SP 800-56B Rev. 2: Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography, 2019. https://doi.org/10.6028/NIST.SP.800-56Br2.

[Nat20] National Institute of Standards and Technology and Canadian Centre for Cyber Security. Implementation guidance for FIPS 140-3 and the cryptographic module validation program, 2020. https://csrc.nist.gov/CSRC/media/Projects/cryptographic-module-validation-program/documents/fips140-3/FIPS140-3IG.pdf (accessed 2023-07-15).

[Nat22a] National Institute for Standards and Technology. Post-quantum cryptography: Additional digital signature schemes, December 2022. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf (accessed 2025-08-02).

[Nat22b] National Security Agency. The commercial national security algorithm suite 2.0 and quantum computing faq, Sep 2022. https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/0/CSI_CNSA_2.0_FAQ_.PDF (accessed 2025-08-02).

[Nat24a] National Institute of Standards and Technology. Fips 203: Module-Lattice-Based module-lattice-based key-encapsulation mechanism standard, 2024.

[Nat24b] National Institute of Standards and Technology. Fips 204: Module-Lattice-Based digital signature standard, 2024.

[Nat24c] National Institute of Standards and Technology. Fips 205: Stateless hash-based digital signature standard, 2024.

[Nora] Nordic Semiconductor. Arm trustzone cryptocell 310. https://docs.nordicsemi.com/bundle/ps_nrf52840/page/cryptocell.html (accessed 2025-01-29).

[Norb] Nordic Semiconductors. Nrf52840 development kit brief. https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52840-DK-product-brief.pdf (accessed 2025-01-29).

[NSS23] NSS Authors. NSS Source Code, 2023. https://hg.mozilla.org/projects/nss/file/b7888f994479307ea70bfbd5c2b1bb4cd6c36a55/lib/softoken/pkcs11c.c (accessed 2023-15-07).

[OBG+24] Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, and Peter Schwabe. High-assurance zeroization. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):375–397, 2024. https://eprint.iacr.org/2023/1713.

[OEKBN+24] Daniel Owens, Rabih El Khatib, Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Efficient and side-channel resistant Ed25519 on ARM Cortex-M4. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[OKS00] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. Elliptic curves with the Montgomery-form and their cryptographic applications. In Hideki Imai and Yuliang Zheng, editors, *Public-Key Cryptography – PKC 2000*, volume 1751 of *LNCS*, pages 238–257. Springer, 2000. https://link.springer.com/content/pdf/10.1007/978-3-540-46588-1_17.pdf.

[Olv25a] Olvid. Olvid – technology: One giant leap for messaging, 2025. https://olvid.io/technology/en/ (accessed 2025-04-14).

[Olv25b] Olvid. Olvid Keycloak configuration guide, 2025. https://www.olvid.io/keycloak/ (accessed 2025-04-08).

[Opea] Open Quantum Safe Project. Open Quantum Safe. https://openquantumsafe.org (accessed 2022-05-20).

[Opeb] OpenSK Authors. Google OpenSK source code. github.com/google/OpenSK. (accessed 2025-01-29).

[Ope23] OpenSSL Authors. OpenSSL Source Code, 2023. https://github.com/openssl/openssl/blob/0e9725bcb90770d967351b977407b174bbd91869/crypto/mem_clr.c (accessed 2023-07-15).

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of

LNCS, pages 1–20. Springer, 2006. http://eprint.iacr.org/2005/271/.

[PAC19] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, 2019.

[Pas09] Sylvain Pasini. *Secure communication using authenticated channels.* PhD thesis, EPFL, Switzerland, 2009. https://secu.famillepasini.ch/files/2009/phd/pasini_phd_thesis.pdf.

[Pat95] Jacques Patarin. Cryptanalysis of the Matsumoto and Imai public key scheme of Eurocrypt'88. volume 963 of *LNCS*, pages 248–261. Springer, 1995.

[Pat96] Jacques Patarin. Hidden fields equations (HFE) and isomorphisms of polynomials (IP): two new families of asymmetric algorithms. volume 1070 of *LNCS*, pages 33–48. Springer, 1996.

[Pat97] Jacques Patarin. The oil and vinegar signature scheme. In *Dagstuhl Workshop on Cryptography*, September 1997.

[PCG01] Jacques Patarin, Nicolas T. Courtois, and Louis Goubin. Quartz, 128-bit long digital signatures. In *CT-RSA*, volume 2020 of *LNCS*, pages 282–297. Springer, 2001.

[Per05] Colin Percival. Cache missing for fun and profit. In *BSDCan '05*. FreeBSD, 2005. https://papers.freebsd.org/2005/cperciva-cache_missing/.

[Per14] Colin Percival. Zeroing buffers is insufficient. Post on on the Daemonic Dispatches blog, 2014. https://www.daemonology.net/blog/2014-09-06-zeroing-buffers-is-insufficient.html (accessed 2023-07-16).

[Per18] Trevor Perrin. Noise protocol framework, 2018. https://noiseprotocol.org/noise.pdf (accessed 2025-08-02).

[PFH+20a] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. available at https://falcon-sign.info/.

[PFH+20b] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin,

Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[PGBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure DSA signing exponentiations really are constant-time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1650. ACM, 2016. https://eprint.iacr.org/2016/594.pdf.

[PGC98] Jacques Patarin, Louis Goubin, and Nicolas T. Courtois. ${C^*}_{-+}$ and HM: variations around two schemes of T. Matsumoto and H. Imai. In *ASIACRYPT*, volume 1514 of *LNCS*, pages 35–49. Springer, 1998.

[PK01] Andreas Pfitzmann and Marit Köhntopp. Anonymity, unobservability, and pseudonymity – a proposal for terminology. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *LNCS*, pages 1–9. Springer, 2001. https://dud.inf.tu-dresden.de/Anon_Terminology.shtml.

[PKLN21] Sebastian Paul, Yulia Kuzovkova, Norman Lahr, and Ruben Niederhagen. Mixed certificate chains for the transition to post-quantum authentication in TLS 1.3. Cryptology ePrint Archive, Report 2021/1447, 2021. https://eprint.iacr.org/2021/1447.

[Por19] Thomas Pornin. New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Report 2019/893, 2019. https://eprint.iacr.org/2019/893.

[Por23] Thomas Pornin. BearSSL API Overview – Memory Wiping, 2023. https://www.bearssl.org/api1.html (accessed 2023-07-13).

[PST23] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from threema: Analysis of a

secure messenger. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1289–1306. USENIX Association, 2023. `https://www.usenix.org/conference/usenixsecurity23/presentation/paterson`.

[PV06] Sylvain Pasini and Serge Vaudenay. SAS-based authenticated key agreement. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *LNCS*, pages 395–409. Springer, 2006. `https://iacr.org/archive/pkc2006/39580401/39580401.pdf`.

[Res18] Eric Rescorla. The Transport Layer Security TLS Protocol Version 1.3. RFC 8446, RFC Editor, August 2018. `https://rfc-editor.org/rfc/rfc8446.txt`.

[RKK20] Johannes Roth, Evangelos G. Karatsiolis, and Juliane Krämer. Classic McEliece implementation with low memory footprint. In *CARDIS*, volume 12609 of *LNCS*, pages 34–49. Springer, 2020.

[RMS18] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 415–429. IEEE, 2018. `https://doi.org/10.1109/EuroSP.2018.00036`.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 98–107. ACM Press, November 2002.

[RPS18] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1397–1414. ACM, 2018. `https://eprint.iacr.org/2018/747.pdf`.

[RS06] Alexander Rostovtsev and Anton Stolbunov. PUBLIC-KEY CRYPTOSYSTEM BASED ON ISOGENIES. Cryptology ePrint Archive, Paper 2006/145, 2006. `https://eprint.iacr.org/2006/145`.

[Rus23a] Rust Authors. Rust's emit-stack-sizes, 2023. `https://doc.rust-lang.org/unstable-book/compiler-flags/emit-stack-sizes.html` (accessed 2023-07-13).

[Rus23b] RustCrypto Project Developers. Zeroize Source Code, 2023. `https://github.com/RustCrypto/utils/tree/master/zeroize` (accessed 2023-07-15).

[Saa24a] Markku-Juhani O Saarinen. Accelerating SLH-DSA by two orders of magnitude with a single hash unit. In *Annual International Cryptology Conference*, pages 276–304. Springer, 2024.

[Saa24b] Markku-Juhani O Saarinen. Accelerating SLH-DSA by two orders of magnitude with a single hash unit. In *Annual International Cryptology Conference*, pages 276–304. Springer, 2024.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`.

[SBB+22] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. Cryptology ePrint Archive, Report 2022/426, 2022. `https://eprint.iacr.org/2022/426`.

[SBG+22] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing high-speed cryptography against spectre v1. Cryptology ePrint Archive, Report 2022/1270, 2022. `https://eprint.iacr.org/2022/1270`.

[SCA18] Laurent Simon, David Chisnall, and Ross Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15, 2018. `https://www.cl.cam.ac.uk/~rja14/Papers/whatyouc.pdf`.

[SCH+19] Simona Samardjiska, Ming-Shing Chen, Andreas Hulsing, Joost Rijneveld, and Peter Schwabe. MQDSS. Technical report, National Institute of Standards and Technology, 2019. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.

[Sho94] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Annual Symposium on Foundations of Computer Science – FOCS 1994*, pages 124–134. IEEE, 1994. `https://ieeexplore.ieee.org/abstract/document/365700`.

[Sho01] Victor Shoup. A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Paper 2001/112, 2001. `https://eprint.iacr.org/2001/112`.

[SKD20] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. Post-quantum authentication in TLS 1.3: A performance study. In *NDSS 2020*. The Internet Society, February 2020.

[SKH+19] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656. USENIX Association, 2019. `https://www.usenix.org/conference/usenixsecurity19/presentation/shusterman`.

[SLP+25] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking bad: How compilers break constant-time implementations. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 1690–1706. ACM, 2025. `https://dl.acm.org/doi/full/10.1145/3708821.3733909`.

[SMCB12] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 78–94. IEEE, 2012. `https://infsec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/dh_tamarin_csf.pdf`.

[Spr22] Amber Sprenkels. Eraser, 2022. `https://github.com/dsprenkels/eraser` (accessed 2023-07-15).

[SSD21] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1019–1035. USENIX Association, 2021. `https://www.usenix.org/conference/usenixsecurity21/presentation/shahverdi`.

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1461–1480, New York, NY, USA, 2020. Association for Computing Machinery.

[SSW21] Peter Schwabe, Douglas Stebila, and Thom Wiggers. More efficient post-quantum KEMTLS with pre-distributed public keys. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021, Part I*, volume 12972 of *LNCS*, pages 3–22. Springer, Heidelberg, October 2021.

[Ste24] Douglas Stebila. Security analysis of the imessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/357, 2024. https://eprint.iacr.org/2024/357.

[Str10] Falko Strenzke. How to implement the public key operations in code-based cryptography on memory-constrained devices. Cryptology ePrint Archive, Report 2010/465, 2010. https://eprint.iacr.org/2010/465.

[Syn14] Synopsys Inc. The Heartbleed Bug, 2014. https://heartbleed.com/ (accessed 2023-15-07).

[Syn20] Synacktiv. Cible de sécurié cspn olvid, version 2.2, 2020. https://cyber.gouv.fr/sites/default/files/2020/10/anssi-cible-cspn-2020_30fr.pdf (accessed 2025-08-02).

[Syn21] Synacktiv. Rapport technique d'évaluation olvid android, version 1.2, 2021. https://olvid.io/assets/documents/Synacktiv-Olvid-CSPN_Olvid-0.9.2-RTE-v1.2.pdf (accessed 2025-08-02).

[TCG+] Noemi Terzo, Cas Cremers, Ruben Gonzalez, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. Security analysis of the olvid secure messenger. *unpublished*.

[Tin23] TinyDTLS Authors. TinyDTLS Source Code, 2023. https://github.com/eclipse/tinydtls/blob/004aba8f7a1f7b70eb1a43dfa9fc4be644daa4ca/crypto.c#L254 (accessed 2023-07-15).

[Toc] Tock Embedded OS. tockos. https://github.com/tock/tock. (accessed 2025-01-29).

[TPD20] Chengdong Tao, Albrecht Petzoldt, and Jintai Ding. Improved key recovery of the HFEv- signature scheme. Cryptology ePrint Archive, Paper 2020/1424, 2020. https://eprint.iacr.org/2020/1424.

[vdLPR+18] Ebo van der Laan, Erik Poll, Joost Rijneveld, Joeri de Ruiter, Peter Schwabe, and Jan Verschuren. Is Java Card ready for hash-based signatures? In *International Workshop on Security*, pages 127–142. Springer, 2018.

[Wei08] Florian Weimer. New openssl packages fix predictable random number generator. Debian Security Advisory 1571-1, 2008. Bug discovered by Luciano Bello, https://lists.debian.org/debian-security-announce/2008/msg00152.html (accessed 2025-08-02).

[Wes21] Bas Westerbaan. Sizing up post-quantum signatures, November 2021. https://blog.cloudflare.com/sizing-up-post-quantum-signatures/ (accessed 2022-06-17).

[Wes23] Bas Westerbaan. Cloudflare now uses post-quantum cryptography to talk to your origin server, 2023. https://blog.cloudflare.com/post-quantum-to-origins/ (accessed 2025-02-03).

[Wol23] WolfSSL Authors. WolfSSL Source Code, 2023. https://github.com/wolfSSL/wolfssl/blob/e2424e67444a360eab615b53fd5649ff355ad68b/wolfcrypt/src/misc.c#L349 (accessed 2023-07-15).

[XLD+23] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *Proceedings of USENIX Security Symposium, 2023*. USENIX, 2023.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732. USENIX Association, 2014. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[YFT20] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020. USENIX Association, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/yan.

[YJO+17] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 1025–1040. USENIX Association, August 2017.

[Zep] Zephyr Project. Zephyr documentation. https://www.zephyrproject.org (accessed 2025-08-02).

Research Data Management

This thesis research has been carried out under the research data management policy of Radboud University, The Netherlands.[1]

The following research datasets, consisting only of source code, have been produced during this PhD research and are available individually on public repositories and via unique DOI `http://doi.org/10.5281/zenodo.17381244`:

- Chapter 3: Verifying Post-Quantum Signatures in 8 kB of RAM. `https://git.fslab.de/pqc/streaming-pq-sigs`
- Chapter 4: Formally Verified Zeroization of Secret Values in RAM. `https://artifacts.formosa-crypto.org/data/clearstack.tar.bz2`
- Chapter 5: KEMTLS vs. Post-Quantum TLS in Resource-Constrained Devices. `https://github.com/rugo/wolfssl-kemtls-experiments/`
- Chapter 6: Post-Quantum FIDO with Hash-Based Signatures. `https://github.com/rugo/fido-sphincs-experiments`
- Chapter 7: Security Analysis of the Olvid Secure Messenger. `https://anonymous.4open.science/api/repo/analysing_olvid/file/olvidgbu.tar.bz2`

[1] `https://www.ru.nl/en/about-us/policies-and-regulations/research-data-management/guidelines-for-research-data-management`, accessed May 27th, 2025.

Summary

This thesis explores the implementation of post-quantum cryptography (PQC) in resource-constrained environments, providing insights into the feasibility and cost of deploying PQC in such settings. Its main focus is on quantum-secure authentication on embedded systems. The chapters presented elaborate on how PQC can be integrated into various embedded scenarios. Such integrations typically require careful consideration of the system's specifics and necessitate vigilant implementation and algorithmic tweaks. An introduction and motivation to the subject is given in Chapter 1. Chapter 2 then gives necessary definitions, details notation and presents the research methodology. The remainder of the thesis is divided into three parts. Part I details research on implementations of PQC primitives in resource-constrained environments and their security. It consists of Chapter 3 and Chapter 4.

Chapter 3. In this chapter, we study implementations of post-quantum signature schemes on resource-constrained devices. We focus on verification of signatures and cover NIST PQC round-3 candidates Dilithium, Falcon, Rainbow, GeMSS, and SPHINCS$^+$. For this, we assume an ARM Cortex-M3 with 8 kB of memory and 8 kB of flash for code; a practical and widely deployed setup in, for example, the automotive sector. This amount of memory is insufficient for most schemes. Rainbow and GeMSS public keys are too big; SPHINCS$^+$ signatures do not fit in this memory. To make signature verification works for these schemes, we stream in public keys and signatures. When streaming the public key, the device needs to securely store a hash value of the public key to verify the authenticity of the streamed public key. During signature verification, the public key is incrementally hashed, matching the data flow of the streamed public key. Due to the memory requirements for efficient Dilithium implementations, we stream in the public key to cache more intermediate results. We discuss the suitability of the signature schemes for streaming, adapt existing implementations, and compare performance.

Chapter 4. This chapter introduces a solution that can benefit implementation of post-quantum, pre-quantum and symmetric primitives alike. In it, we revisit the problem of erasing sensitive data from memory and registers during return from a cryptographic routine. While the problem

and related attacker model is fairly easy to phrase, it turns out to be surprisingly hard to guarantee security in this model when implementing cryptography in common languages such as C/C++ or Rust. We revisit the issues surrounding zeroization and then present a principled solution in the sense that it guarantees that sensitive data is erased and it clearly defines when this happens. We implement our solution as extension to the formally verified Jasmin compiler and extend the correctness proof of the compiler to cover zeroization. We show that the approach seamlessly integrates with state-of-the-art protections against microarchitectural attacks by integrating zeroization into Libjade, a cryptographic library written in Jasmin with systematic protections against timing and certain microarchitectural attacks. We present benchmarks showing that in many cases the overhead of zeroization is barely measurable.

Part II, containing Chapter 5 and Chapter 6, builds up on this research and investigates how security protocols used in embedded environments can be altered and instantiated to be quantum secure, while retaining necessary performance characteristics.

Chapter 5. In this chapter, we conduct a performance comparison of the established TLS 1.3 and the newly proposed KEMTLS in an embedded setting, including different transmission mediums. The comparison has a focus on server authentication within the embedded client, but takes the full post-quantum handshake into account. TLS secures transport for high-end desktops and low-end embedded devices alike. With the first PQC standards around, TLS will have to be updated soon. However, especially for small microcontrollers, it appears the current NIST-selected post-quantum signature solutions pose a challenge. To gain meaningful results, this chapter presents implementations of KEMTLS and TLS 1.3 on a Cortex-M4-based platform. In the experiments, both protocols are investigated with the NIST finalist signature schemes and KEMs, except for Classic McEliece which has too large public keys. All benchmarks are performed in network settings relevant to the Internet of Things, namely low-latency broadband, LTE-M and Narrowband IoT.

Chapter 6. This chapter continues to investigate post-quantum authentication in resource-constrained environments. In contrast to the previous chapter, this chapter is concerned with user authentication. FIDO2 can be used to facilitate secure user authentication within an application that provides server authentication (as seen in the previous chapter). It leverages asymmetric signatures for that purpose. Two of the three NIST-selected signature algorithms are lattice based, offering good performance but posing challenges due to complex implementation and intricate security assumptions. A more conservative choice for quantum-safe authentication are hash-based signature systems, such as the also-standardized SLH-DSA, which is based on the SPHINCS+ construction. However, due

to large signature sizes and low signing speeds, hash-based systems have only found use in niche applications. In this chapter we combine different approaches to show that the SPHINCS+ signature system can be optimized in its parameters and implementation to be high performing, even when signing in an embedded setting. We demonstrate this in the context of user authentication using hardware security keys within FIDO.

Part III, containing Chapter 7 features supplementary work that is relevant also outside the embedded and post-quantum realm.

Chapter 7. In this chapter, we present an analysis of the cryptographic core of the French encrypted messaging app Olvid. It is, among other things, a recommended instant messaging solution for french government officials. We employ the TAMARIN prover for a formal analysis of Olvid's authenticated key-exchange and continuous-key-agreement protocols, and conduct an analysis of the Java implementation targeting Android devices. Even though the formal analysis confirms that the Olvid protocols achieve security properties such as mutual authentication, session-key secrecy, forward secrecy, and replay protection, the analysis also reveals that the Olvid protocols fail to achieve security under certain key-reveal patterns. Furthermore, source-code analysis reveals at least one timing leakage vulnerability, for which we present a practical key-recovery attack. To enhance Olvid's security and facilitate future analyses, we propose design changes aimed at mitigating the identified vulnerabilities.

All chapters are accompanied by code published under a permissive license, deliver extensive benchmarks and present reproducible results.

Samenvatting

Dit proefschrift onderzoekt de implementatie van post-quantum cryptografie (PQC) in omgevingen met beperkte middelen en geeft inzicht in de haalbaarheid en de kosten van het inzetten van PQC in dergelijke omgevingen. De meeste focus ligt op quantum-veilige authenticatie op *embedded systems*. De gepresenteerde hoofdstukken belichten hoe PQC kan worden geïntegreerd in verschillende embedded scenario's. Dergelijke integraties vereisen doorgaans zorgvuldige overweging van de specificaties van het systeem en maken een waakzame implementatie en algoritmische aanpassingen noodzakelijk. Een introductie en motivatie voor het onderwerp wordt gegeven in Chapter 1. Chapter 2 geeft vervolgens de nodige definities, detaileert notatie en presenteert de onderzoeksmethodologie. De rest van het proefschrift is verdeeld in twee delen.

Part I beschrijft het onderzoek naar implementaties van PQC *primitives* in omgevingen met beperkte middelen en hun beveiliging. Het bestaat uit Chapter 3 en Chapter 4.

Chapter 3. In dit hoofdstuk bestuderen we implementaties van post-quantum digitale handtekeningen op apparaten met beperkte middelen. We richten ons op de verificatie van handtekeningen en behandelen NIST PQC ronde-3 kandidaten Dilithium, Falcon, Rainbow, GeMSS en SPHINCS$^+$. Hiervoor gaan we uit van een ARM Cortex-M3 met 8 kB geheugen en 8 kB flash voor code; een praktische en veelgebruikte opstelling in bijvoorbeeld de auto-industrie. Deze hoeveelheid geheugen is onvoldoende voor de meeste cryptosystemen. De publieke sleutels van Rainbow en GeMSS zijn te groot; SPHINCS$^+$-handtekeningen passen niet in dit geheugen. Om verificatie voor deze cryptosystemen te laten werken, streamen we publieke sleutels en handtekeningen. Bij het streamen van de publieke sleutel moet het apparaat een hashwaarde van de publieke sleutel veilig opslaan om de authenticiteit van de gestreamde publieke sleutel te verifiëren. Tijdens de verificatie wordt de publieke sleutel incrementeel gehasht, om te matchen met de dataflow van de gestreamde publieke sleutel. Vanwege de geheugeneisen voor efficiënte Dilithium-implementaties streamen we de publieke sleutel in om meer tussenresultaten te cachen. We bespreken de geschiktheid van de cryptosystemen voor streaming, passen bestaande implementaties aan en vergelijken prestaties.

Chapter 4. Dit hoofdstuk introduceert een oplossing die de implementatie van post-quantum, pre-quantum en symmetrische *primitives* ten goede kan komen. In deze oplossing kijken we opnieuw naar het probleem van het wissen van gevoelige gegevens uit het geheugen en registers tijdens de return van een cryptografische routine. Hoewel het probleem en het gerelateerde aanvallersmodel vrij eenvoudig te formuleren zijn, blijkt het verrassend moeilijk om veiligheid in dit model te garanderen wanneer we cryptografie implementeren in gangbare talen zoals C/C++ of Rust. We bekijken nogmaals de problemen rond *zeroization* en presenteren vervolgens een oplossing gebaseerd op principes, in de zin dat het garandeert dat gevoelige gegevens worden gewist en het duidelijk definieert wanneer dit gebeurt. We implementeren onze oplossing als uitbreiding op de formeel geverifieerde Jasmin-compiler en breiden het correctheidsbewijs van de compiler uit om *zeroization* te dekken. We tonen aan dat de aanpak naadloos integreert met geavanceerde beschermingen tegen microarchitecturale aanvallen door *zeroization* te integreren in Libjade, een cryptografische bibliotheek geschreven in Jasmin met systematische beschermingen tegen timing en bepaalde microarchitecturale aanvallen. We presenteren benchmarks die aantonen dat de overhead van *zeroization* in veel gevallen nauwelijks meetbaar is.

Part II, welke bestaat uit Chapter 5 en Chapter 6, bouwt voort op dit onderzoek en onderzoekt hoe beveiligingsprotocollen die in embedded omgevingen worden gebruikt, kunnen worden aangepast en geïmplementeerd om quantumveilig te zijn, terwijl de noodzakelijke prestatiekenmerken behouden blijven.

Chapter 5. In dit hoofdstuk voeren we een vergelijking uit tussen de prestatie van de gevestigde TLS 1.3 en de nieuw voorgestelde KEMTLS in een embedded omgeving, voor verschillende transmissiemedia. De vergelijking richt zich op serverauthenticatie binnen de embedded client, maar houdt rekening met de volledige post-quantum handshake. TLS beveiligt transport voor zowel high-end desktops als low-end embedded apparaten. Met de eerste PQC-standaarden in aantocht, zal TLS binnenkort moeten worden bijgewerkt. Echter, vooral voor kleine microcontrollers lijken de huidige door NIST geselecteerde post-quantum digitale handtekeningen een uitdaging te vormen. Om zinvolle resultaten te krijgen, presenteert dit hoofdstuk implementaties van KEMTLS en TLS 1.3 op een Cortex-M4 platform. In de experimenten worden beide protocollen onderzocht met de NIST-finalisten voor digitale handtekeningen en KEM's, behalve Classic McEliece, die te grote publieke sleutels heeft. Alle benchmarks worden uitgevoerd in netwerkinstellingen die relevant zijn voor het Internet of Things, namelijk low-latency breedband, LTE-M en Narrowband IoT.

Chapter 6. Dit hoofdstuk blijft post-quantum authenticatie in omgevingen met beperkte middelen onderzoeken. In tegenstelling tot het vorige

hoofdstuk richt dit hoofdstuk zich op gebruikersauthenticatie. FIDO2 kan worden gebruikt om veilige gebruikersauthenticatie te vergemakkelijken binnen een applicatie die serverauthenticatie biedt (zoals te zien in het vorige hoofdstuk). Voor dat doel maakt het gebruik van asymmetrische handtekeningen. Twee van de drie door NIST geselecteerde algoritmes voor digitale handtekeningen zijn gebaseerd op roosters, waardoor ze goede prestaties bieden maar ook uitdagingen vormen vanwege complexe implementaties en ingewikkelde beveiligingsaannames. Een conservatievere keuze voor quantum-veilige authenticatie zijn digitale handtekeningen gebaseerd op hashfuncties, zoals het eveneens gestandaardiseerde SLH-DSA, dat is gebaseerd op de SPHINCS+ constructie. Echter, vanwege hun grote handtekeningen en trage ondertekeningen, hebben systemen gebaseerd op hashfuncties alleen gebruik gevonden in nichetoepassingen. In dit hoofdstuk combineren we verschillende benaderingen om te laten zien dat het SPHINCS+ cryptosysteem kan worden geoptimaliseerd in zijn parameters en implementatie om hoge prestaties te leveren, zelfs bij het ondertekenen in een embedded omgeving. We demonstreren dit in de context van gebruikersauthenticatie met behulp van hardwarebeveiligingssleutels binnen FIDO.

Part III, welke bestaat uit Chapter 7, bevat aanvullend werk dat ook buiten het embedded en post-quantum domein relevant is.

Chapter 7. In dit hoofdstuk presenteren we een analyse van de cryptografische kern van de Franse versleutelde berichtenapp Olvid. Het is onder andere een aanbevolen oplossing voor *instant messaging* voor Franse overheidsfunctionarissen. We gebruiken de Tamarin *prover* voor een formele analyse van Olvid's geauthenticeerde sleuteluitwisseling en continuesleutelovereenkomstprotocollen, en voeren een analyse uit van de Java-implementatie gericht op Android-apparaten. Hoewel de formele analyse bevestigt dat de Olvid-protocollen beveiligingseigenschappen bereiken zoals wederzijdse authenticatie, sessiesleutelgeheimhouding, *forward secrecy* en replaybescherming, onthult de analyse ook dat de Olvid-protocollen niet in staat zijn om beveiliging te bereiken onder bepaalde patronen van sleutelonthulling. Bovendien onthult broncode-analyse ten minste één kwetsbaarheid voor timing aanvallen, waarvoor we een praktische sleutelherstel-aanval presenteren. Om de beveiliging van Olvid te verbeteren en toekomstige analyses te vergemakkelijken, stellen we ontwerpwijzigingen voor die gericht zijn op het verminderen van de geïdentificeerde kwetsbaarheden.

Alle hoofdstukken worden vergezeld van code die onder een *permissive* licentie is gepubliceerd, leveren uitgebreide benchmarks en presenteren reproduceerbare resultaten.