QuantumRISC

Work Package 3, Deliverables 3.1 and 3.2: Development of Software Libraries

Software Implementation Approaches for Post-quantum Cryptography on Embedded Systems

Version	1.0
Project Coordination	Fraunhofer Institute for Secure Information Technology
Date of preparation	May 5, 2023

SPONSORED BY THE

Federal Ministry of Education and Research

Authors

- Elektrobit Automotive GmbH:
    - Hannes Hennig
- Fraunhofer SIT:
    - Julian Wälde
    - Norman Lahr
    - Richard Petri
- Hochschule RheinMain:
    - Thorsten Knoll
- MTG AG:
    - Evangelos Karatsiolis
    - Johannes Roth
- Ruhr-Universität Bochum:
    - Georg Land
    - Jan Philipp Thoma
- Universität Regensburg:
    - Michael Meyer

Project Coordination
Norman Lahr
Fraunhofer Institute for Secure Information Technology
Advanced Cryptographic Engineering
Rheinstr. 75
D-64295 Darmstadt
Germany

Phone	+49 6151 869100
Fax	+49 6151 869224
Mail	norman.lahr@sit.fraunhofer.de

Contents

1 Executive Summary

QuantumRISC is a project funded by the German Federal Ministry of Education and Research. The goal of the project is to research, analyze, and develop schemes, algorithms, and methods for the secure and efficient use of post-quantum cryptographic algorithms for embedded devices. The project is divided into several work packages. Work package 3 (WP3) of the QuantumRISC project focuses on the design and implementation of post-quantum algorithms and protocols in software. The document at hand presents the work of the partners of the project team regarding WP3.

First we present previous work in this area by giving an overview of existing software libraries and products and briefly discussing them. To easily use and integrate cryptographic schemes in libraries and applications, a flexible and agile API is necessary. We address this issue by specifying an API and analysing its use.

Several post-quantum cryptographic algorithms feature large public-key or signature sizes. We discuss methods and algorithms to mitigate the memory requirements of the cryptographic schemes Classic McEliece and SPHINCS$^+$. We also present a method to accelerate the cryptographic operations of the key encapsulation algorithm BIKE.

Several cryptographic protocols and specifications like TLS and X.509 certificates are described and are evaluated for their use with post-quantum algorithms. We discuss the challenges and propose solutions for a suitable integration for some of these algorithms.

We also analyze and experiment with using more than one cryptographic algorithm within a certificate chain hierarchy. A test suite for the TLS use case is described and the test results are presented.

A very important aspect of implementing cryptographic algorithms in software is their resistance to side-channel attacks such as timing or power analysis attacks. The rest of the document deals with this aspect. A brief overview of side-channel attacks is given. Our implemented countermeasures for post-quantum algorithms are shown. In particular, the constant-time implementation of Classic McEliece and the stateful signature method XMSS in a hardware security module are presented. Lastly, we present a timing attack on the HQC and BIKE schemes.

2 Introduction

This document reports the results of work package 3 (WP3) of the QuantumRISC project. These results are new software methods and algorithms for post quantum algorithms that have been analysed in the project (see [MM22]) and are suited for the specified use cases (see [Noa+20]).

This work package WP3 has two deliverables:

- D 3.1 – Documentation of the software library
- D 3.2 – Documentation of the hardening measures

D 3.1 is documented in Chapter 3 and D 3.2 in Chapter 4. These two deliverables are placed into one report to provide a better overview on the content of the deliverables and present the dependencies among them clearly.

Chapter 3 documents various aspects of the software methods and libraries for post-quantum cryptography developed in this project.

Chapter 4 gives an overview of considered side-channel attacks and countermeasures. The countermeasures contained in the software libraries are discussed and described in detail.

3 D3.1 – Design and Implementation in Software

This chapter presents the results of the research and development of software implementations and optimizations of PQC that are part of the work package WP3.

Section 3.1 gives an overview of previous work with a listing of existing libraries and software implementations of PQC algorithms. Section 3.2 presents the developed API which allows easy integration and use of post-quantum algorithms. Section 3.3 presents various implementations of post-quantum algorithms in software. Optimizations for the algorithms Classic McEliece, Bike, Kyber, and NewHope are shown. Section 3.4 describes the challenges in the use of post-quantum algorithms in various protocols and suggests solutions for some of these cases. Lastly, in Section 3.5 the mixing of PQC schemes in certificate chains is discussed.

3.1 PQC Libraries: Existing and related

The topic of Post-Quantum Cryptography (PQC) is around for already more than a decade now. A potpourri of publications, projects, algorithms, implementations, standardization efforts, and many more interests have arisen since then. The American National Institute of Standards and Technology (NIST) started a process for standardization of PQC, being near the finish line. Ahead of the finalization of the NIST process, the German Federal Office for Information Security (BSI) started giving out first recommendations for practical implementation and usage of PQC. Therefore it is necessary to look into what already has happened and what got developed in the field of software for PQC.

In this section we give an overview about these works, with a focus on existing and lively maintained PQC libraries. Each library gets it's own profile card to display the main characteristics, criteria, and features of it. Appending to the profile cards a list of works related to PQC libraries is provided. Commercial libraries are discussed at the end of this overview. All information about the libraries and related works is collected from their public available websites and dates of the retrieval are given.

Name: LibOQS	**License:** Main: MIT License. External parts: Various
Description: liboqs is an open source C library for quantum-safe cryptographic algorithms. liboqs provides a collection of open-source implementations of quantum-safe key encapsulation mechanisms (KEM) and digital signature algorithms, a common API for these algorithms, a test harness and benchmarking routines.	
Source of Information, Website, 21.11.22: `https://openquantumsafe.org/liboqs/`	
Algorithms: KEMs and Signature schemes from NIST Round 3. Finalists and alternate candidates.	**Coding languages::** C with wrappers in C++, Go, Java, .Net, Python, and Rust.
API: C API	**Platforms** No specific platforms as targets.

Table 3.1: Profile card: liboqs, Source: Website, see link above.

Name: PQClean	
License: No general license. All subdirectories contain LICENSE files or information at the top of single files. Most files are public domain, MIT or CC0.	
Description: PQClean, in short, is an effort to collect clean implementations of the post-quantum schemes that are in the NIST post-quantum project, where clean means that the C code conforms to a defined code quality standard. The goal of PQClean is to provide standalone implementations. The project's website describes these goals in deeper detail and points out what PQClean does not aim at.	
Source of Information, Website, 21.11.22: `https://github.com/PQClean/PQClean`	
Algorithms: Selection from the NIST PQC.	**Coding languages:** C
API: Aims for easy integration into other libraries like liboqs, pqm4, SUPERCOP, or Open Quantum Safe.	**Platforms:** No specific platforms as targets.

Table 3.2: Profile card: PQClean, Source: Website, see link above.

Name: mupq
License: Different subpojects of mupq have different licenses. Individually given in LICENSE files or at the top of single files. Most files are public domain, MIT or CC0.
Description: mupq aims at PQC for microcontrollers (MCU) and contains multiple sub-projects for different platforms (pqm4, pqm3, pqriscv, pqhw, pqriscv). Initial parts of mupq were started out of the EU funded PQCRYPTO project and developed from there into a larger maintainer base. PQC NIST candidates from all rounds are to be found in the subprojects.
Source of Information, Website, 21.11.22: `https://github.com/mupq`

Algorithms: Several algorithms from all NIST rounds: candidates, finalists, and alternatives.	**Coding languages:** C. Some parts (pqhw, pqriscv) parts are written in hardware description languages (HDL).
API: Large parts of the mupq (pqm4) library uses the NIST/SUPERCOP/ PQClean API.	**Platforms:** ARM Cortex M3 and M4, Risc-V, FPGAs, ARM Neon

Table 3.3: Profile card: mupq, Source: Website, see link above.

Name: libpqcrypto	**License:** The components of libpqcrypto vary in licenses. Some parts are in the public domain, but others are not.

Description: libpqcrypto is a new cryptographic software library produced by the PQCRYPTO project.
Source of Information, Website, 21.11.22: `https://libpqcrypto.org/`

Algorithms: PQCRYPTO submissions for NIST PQC	**Coding languages:** C with Python API/wrapper
API: Unified Interfaces in C and Python. Command line tools.	**Platforms:** No specific platforms as targets.

Table 3.4: Profile card: libpqcrypto, Source: Website, see link above.

Name: CIRCL	**License:** Cloudflare open-source license.
Description: CIRCL (Cloudflare Interoperable, Reusable Cryptographic Library) is a collection of cryptographic primitives written in Go. The goal of this library is to be used as a tool for experimental deployment of cryptographic algorithms targeting PQC and ECC.	
Source of Information, Website, 21.11.22: `https://github.com/cloudflare/circl`	
Algorithms: PQC and ECC	**Coding languages:** Go
API: No specific API.	**Platforms:** No specific platforms as targets.

Table 3.5: Profile card: CIRCL, Source: Website, see link above.

Name: NIST PQC standardization process	**License:** Various licenses.
Description: The complete collection of all submissions for the NIST Post-Quantum Cryptography Standardization Process. This includes the history of the submissions over all four rounds.	
Source of Information, Website, 21.11.22: `https://csrc.nist.gov/projects/post-quantum-cryptography`	
Algorithms: All submissions. All Algorithms.	**Coding languages:** C, C++
API: NIST PQC C header API.	**Platforms:** No specific platforms as targets.

Table 3.6: Profile card: NIST PQC References, Source: Website, see link above.

Related to PQC libraries: Not beeing full PQC libraries in itself, but very close related to the mentioned libraries are the following works.

- SUPERCOP: A toolkit for measuring the performance of cryptographic software, widley used and implemented by libraries.[1]
- MbedTLS implements cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and DTLS protocols.[2]
- BOTAN is a C++ cryptography library released under the permissive Simplified BSD license.[3]

[1] https://bench.cr.yp.to/supercop.html
[2] https://github.com/Mbed-TLS/mbedtls
[3] https://botan.randombit.net/

- Bouncy Castle is a Java cryptographic library, with several PQC implementations.[4]

Commercial PQC libraries:

- PQSLib describes itself as a Lightweight PQC library for Embedded, IoT, and Secure Elemets.[5]
- Radiate Toolkit describes itself as a high-performance, lightweight, standards-based quantum-safe software development kit.[6]

3.2 Proposed PQC API

The first step to achieve an agile API for a set of cryptographic schemes is to provide a standardized interface. In the past, this first step was a major hurdle, as neither programming interfaces nor data formats were sufficiently defined and varied between cryptographic schemes. In the case of asymmetric cryptographic schemes, the common approach nowadays is to abstract the differences behind a standardized interface.

In our agile API, key generation for all asymmetric schemes is handled by a function that generates a key pair and stores it in a byte buffer of an implementation defined size:

```
int {kem,sign}_keypair(byte* pk, byte* sk);
```

Note that the contents of the byte buffer are opaque to the programmer. PQC schemes usually explicitly define a storage format for the keys and implementations adhere to it, except for the secret key, which may in some cases be implementation defined, since it is usually never intended to be exchanged and used by another implementation. In either case, the programmer using such an API does not need to be concerned with storage formats except for byte arrays, albeit of different sizes. Similar concepts apply to the key encapsulation and decapsulation or signature generation and verification.

With such a standardized interface, the process of exchanging a cryptographic scheme in the application is as easy as exchanging a library against which the application is compiled. If, however, an application must support multiple schemes at the same time, this approach falls short. To facilitate this, we developed two mechanisms that enable an application to support any number of schemes at the same time.

1. *Compile time agility:* The first approach utilizes a strict naming convention for functions and a set of macros, as shown in an excerpt from the API in Listing 3.1. All functions for a specific cipher have a suffix with the name defined for a cipher. The macros starting at line 12 then help a programmer to select the correct function in a flexible manner. An example is shown in Listing 3.2. The first parameter to these macros specifies the cipher and the macro will resolve to the

[4] `https://www.bouncycastle.org/`
[5] `https://pqshield.com/solutions/pqslib/`
[6] `https://www.isara.com/products/isara-radiate.html`

Listing 3.1: Excerpt of function names and macros for compile time agility.

```
typedef enum cbb_kem_id {
  KYBER512 = 4,
  SABER = 9,
} cbb_kem_id_t;

int cbb_kem_genkeypair_KYBER512(/* ... */);
int cbb_kem_genkeypair_SABER(/* ... */);
int cbb_kem_encapsulate_KYBER512(/* ... */);
int cbb_kem_encapsulate_SABER(/* ... */);

#define _CAT(A, B) A##B
#define CBB_KEM_GENKEYPAIR(CIPHER, ...) \
_CAT(cbb_kem_genkeypair_, CIPHER)(__VA_ARGS__)
#define CBB_KEM_ENCAPSULATE(CIPHER, ...) \
_CAT(cbb_kem_encapsulate_, CIPHER)(__VA_ARGS__)
```

Listing 3.2: Program code using compile time agility.

```
#define MYCIPHER1 KYBER512
#define MYCIPHER2 SABER

unsigned char sk1[/*...*/], pk1[/*...*/];

void function() {
  CBB_KEM_GENKEYPAIR(MYCIPHER1, pk1, sk1);
  /* ... */
}
```

Listing 3.3: Excerpt of dynamic functions for run time agility.

```c
typedef enum cbb_kem_id {
  KYBER512 = 4,
  SABER = 9,
} cbb_kem_id_t;

int cbb_kem_genkeypair(cbb_kem_id_t cipher, /* ... */);
int cbb_kem_encapsulate(cbb_kem_id_t cipher, /* ... */);
int cbb_kem_decapsulate(cbb_kem_id_t cipher, /* ... */);
```

Listing 3.4: Program code using run time agility.

```c
void function(cbb_kem_id_t cipher) {
  cbb_kem_encapsulate(cipher, /* ... */);
  /* ... */
}
```

function of the correct name. Note that the name of the cipher may be defined by macro itself, which enables a programmer to change a cipher simply by redefining a macro.

2. *Run time agility:* The first approach works well if the used cipher is fixed for an application, which may be the case for many constrained applications. It does, however, fall short if an application does not know which cipher is to be used ahead of time. Therefore, the API also features a set of dynamic functions as shown in Listing 3.3. An application can then pass a variable that specifies the cipher to these functions, as shown in Listing 3.4, that will map the call to the appropriate cipher function.

The first, static approach is the most suitable for resource constrained applications that only require support for one scheme. It enables a programmer to create an application in a manner that is independent from the cryptographic scheme in use, allowing a quick exchange of schemes if necessary. Furthermore, modern compiler toolchains will automatically remove the code of the unused ciphers, as their symbols are never referenced. The second, dynamic approach comes with the additional cost of a small abstraction layer that multiplexes between all supported ciphers. This abstraction layer will then be a reference to all compiled ciphers, making automatic code removal impossible. To alleviate this, our build system is capable of only including a limited set of ciphers, if only few a required.

Both compile and run time agility concepts require the build system to combine all required cryptographic scheme implementations into a single library, which may lead to problems if some implementations share certain symbols. To be able to easily extend the library in the future, the build system was extended to perform symbol renaming. Each cryptographic scheme implementation is first compiled separately. The symbols defined by each implementation are then listed and renamed, i.e., the build system will automatically append the name of the cipher to each symbol. Note that this only affects symbols defined by the implementations, and not external symbols used by the libraries.

Since the description of a flexible API that also integrates hardware implementations overlaps with work package WP4, parts of this section are also described in [Qrw].

3.3 Software Implementation and Optimization of PQC Algorithms

From the perspective of software implementations, post-quantum schemes significantly differ from their pre-quantum counterparts. For instance, RSA only requires simple modular arithmetic for large integers, and elliptic curve cryptography (ECC) uses the relatively simple elliptic curve group law, which in turn requires efficient finite field arithmetic. However, for post-quantum schemes this usually differs significantly, since they are based on different mathematical hardness assumptions. In the following, we briefly describe which routines are required for post-quantum schemes. For a more general overview of PQC schemes and their applicability, we refer to the QuantumRISC work package 2 [MM22].

Code-based cryptography usually requires mostly matrix multiplications and matrix-vector multiplications. Further, most schemes use binary fields as underlying structure. Thus, it not only differs from pre-quantum schemes from a mathematical perspective, but also regarding the subroutines required for software implementations.

Isogeny-based cryptography is related to elliptic curve cryptography, i.e., it reuses many subroutines such as the elliptic curve group law and arithmetic over finite fields. However, on a higher level, the computations of isogenies add significant complexity to implementations.

Lattice-based cryptography spends most of its computational effort on polynomial arithmetic. Thus, implementations usually optimize the choice of parameters such that polynomial multiplication is very efficient. Implementations often use Number Theoretic Transform (NTT) multiplications to achieve this goal.

Multivariate cryptography resembles code-based and lattice-based schemes in the sense that it usually requires matrix-vector multiplications. Furthermore, most schemes make use of Gauss elimination. As for code-based schemes, mostly binary fields are used.

Hash-based cryptography is an exception in the realm of PQC, in the sense that most of its computational cost is spent on evaluating hash functions. This is a well-known primitive from pre-quantum cryptography, such that existing optimized implementations of hash functions can be reused for schemes like SPHINCS$^+$, LMS, or XMSS.

By using streaming in computations of several post-quantum algorithms, these algorithms can be optimized, especially in the case of embedded devices. This is described in Section 3.3.1. Speed optimization for the key encapsulation mechanism BIKE are shown in Section 3.3.2. Optimizations in the implementation of the polynomial arithmetic used in the algorithms Kyber and NewHope are presented in Sections 3.3.3, 3.3.4, and 3.3.5.

3.3.1 Mitigation of Memory Requirements with a Streaming Approach

Post-quantum schemes can have large signatures, ciphertexts, or keys. For example, Classic McEliece public keys range from roughly a quarter of a megabyte to over a megabyte, and SPHINCS$^+$ signatures are up to 49 kilobytes. The size alone can prevent the use of some post-quantum schemes in low-memory microcontrollers that usually only have memory between 8 to 256 kilobytes. To mitigate the memory requirements, a streaming approach was explored. The term *streaming* in this context means that the data is not processed as one large block of memory but rather processed in smaller chunks at a time. This is analogous to a video stream, where only the current frame (and possibly some past and future frames) are needed at one point in time. There is no need to load the complete video file into the device's memory. In the context of post-quantum algorithms this means that only a small chunk of a key or ciphertext/signature is processed at a time. One goal is to implement streaming in a transparent way, i.e., when viewed from the outside, the cryptographic operation should not behave differently and no protocol changes shall be necessary to support streaming.

This kind of memory-optimization is only applicable if the following is satisfied:

1. The cryptographic operation is performed in an on-line setting. Since the data in question, i.e., a key or ciphertext/signature, cannot be stored on the device's memory due to size constraints, there needs to be some sort of on-line protocol in which the data is sent to or received from. This is satisfied by any cryptographically secured, online communication protocol where some sort of key exchange based on asymmetric cryptography is performed. This can however also be satisfied by other forms of communication, e.g., a signature could be streamed to (or from) the device's own flash memory.

2. The cryptographic primitive can be adapted in such a way that the data can be processed in distinct chunks without the need to process an already processed chunk again. The restriction to this case makes sense since otherwise transparency cannot be achieved.

A more abstract view of the streaming approach is that the communication channel serves as a special kind of buffer. It is a buffer where large cryptographic data is read from or written to with the limitation that it can only either be read from sequentially or written to sequentially and there is no way to go back to the beginning of the buffer. The upside is that conceptually, the buffer is not stored on the device itself, but in the communication layer. In more practical terms, usually some parts would be buffered in the I/O layer until it decides that enough data is accumulated to be read from the application on the device or to be transmitted to the other communication party(s).

The effectiveness of applying the streaming approach to a cryptographic scheme depends on how memory-efficiently one can adapt the scheme to use this special model of a buffer.

The approach was explored and implemented for the Classic McEliece public keys and the SPHINCS$^+$ signatures and enables the use of these schemes with less memory than would otherwise be required. In the following, the changes to the schemes are described. A more detailed description and evaluation can be found in the corresponding published papers [NRW21], [RKK21]. Furthermore, the adapted code that enables streaming for the schemes can be found on Github[7,8].

Streaming of Classic McEliece Public Keys

The Classic McEliece scheme is often deemed not suitable for low-memory devices because of its large public keys. Table 3.7 depicts the Classic McEliece public key sizes.

Parameter Set	n	k	Public Key Size
mceliece348864	3488	2720	261120
mceliece460896	4608	3360	524160
mceliece6688128	6688	5024	1044992
mceliece6960119	6960	5413	1047319
mceliece8192128	8192	6528	1357824

Table 3.7: Classic McEliece public key sizes in bytes and the parameters n and k. k is a parameter that is derived from the binary Goppa code parameters and together with n it determines the public key dimensions.

Classic McEliece is a code-based key encapsulation mechanism scheme. It is in essence the Niederreiter variant of the original scheme that was proposed by Robert McEliece in 1978 and uses binary Goppa codes. The OW-CPA secure Niederreiter PKE is transformed to an IND-CCA2 secure KEM. The private key is a randomly chosen binary Goppa code (in the form of a generating polynomial) and the public key is the non-trivial part of a specific parity-check matrix for the linear code. This parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ is in systematic form, i.e., $H = (I \mid T)$ with $I \in \mathbb{F}_2^{(n-k)\times(n-k)}$ an identity matrix and $T \in \mathbb{F}_2^{(n-k)\times k}$ the public key.

The public key plays a role in the key pair generation operation, as well as in the encapsulation operation. Both cases require a different approach to enable streaming.

The mceliece6960119 parameter set has been omitted from the streaming approach since the parameters produce bit sequences of lengths that are not multiples of 8. However, modern platforms operate on bytes. The trailing bits need an extra handling which the current implementation does not consider. This has no influence on the results.

Streaming Encapsulation. To compute the encrypted shared secret, the encapsulation operation makes use of the public key and computes the syndrome $s = He$ where e is a randomly chosen column vector. This is not the complete encapsulation operation, but the only part where the public key plays a role and hence the part where the streaming is applied.

A typical implementation would implement the matrix-vector product by holding H and e in a buffer and then perform the multiplication. For implementing the streaming approach, a possibility is to read in the public key in single rows or columns and

[7] `https://github.com/MTG-AG/streamingCME`
[8] `https://github.com/QuantumRISC/mbedSPHINCSplusArtifact`

compute $s = He = Ie + Te$ by updating the intermediate result as each row or column is read. The first step is to compute $Ie = e$ which is independent of the public key and amounts to initializing the resulting vector s with e. Then, as the rows or columns of the public key are received, the term Te can be partially computed for each row or column. This is achieved by simply multiplying the received part of the public key by e and then adding it to the intermediate value of s. After all rows or columns have been processed, s holds the result of the product He.

It shall be noted that setting the size of the chunks to one row or column of the public key is an arbitrary choice. As all terms in the product Te are independent, any amount of bytes of the public key in any order can be consumed in one step. Choosing the granularity of rows or columns, however, already reduces the memory requirements for the complete encapsulation operation to under 3 kilobytes for all parameter sets.

Utilizing the streaming approach for the encapsulation operation comes at no additional cost since no computation needs to be carried out that would not be carried out by a non-streaming approach. This makes the public operation of the Classic McEliece scheme appealing to embedded clients. As already mentioned, the streaming approach can also be used for internal communication in the device, e.g., the encapsulation can be performed with a public key that is stored on the device's flash memory without loading it into the (possibly too small) main memory.

Key Pair Generation and Streaming. While the private key holder does not actually need the public key to perform any cryptographic operations, the public key does need to be sent to the other communication participant(s). Usually, a key generation would simply write the public and the private key to a buffer in memory but with the large public keys this can be impossible. One solution is to compute small chunks of the public key at a time, e.g., single rows or single columns. As each chunk is computed, it could be sent to the other participant(s) and then the next chunk is computed. It is straightforward to compute the Classic McEliece public key from the private key, however, doing this in small chunks at a time is more challenging. In the Classic McEliece specification, first some parity-check matrix $\hat{H}$ is produced for which the rows or columns can be computed memory-efficiently, i.e., within a few kilobytes. However, to obtain the public key T, the parity-check matrix is transformed to the systematic form $H = (I \mid T)$. The specification suggests to use the Gaussian elimination algorithm to achieve this. Applying Gaussian elimination requires to hold the complete matrix $\hat{H}$ in memory, where $\hat{H}$ is larger than the public key itself. For streaming single rows or columns, a more memory-efficient approach is required.

A solution is to compute the unique matrix $S \in \mathbb{F}_2^{(n-k)\times(n-k)}$ for which $S\hat{H} = H$. This matrix S is smaller than the public key by a factor of $k/(n-k)$. It can be added to the private key which increases its size. In exchange, the public key does not need to be stored in memory. Instead, single rows or columns of H, and thus T, can be computed by computing the respective row or column of $\hat{H}$ and subsequently computing the product of S by the row or column of $\hat{H}$. The matrix S can be computed as the inverse of S^{-1} which is the leftmost $(n-k) \times (n-k)$ matrix of $\hat{H}$. Computing the inverse via the Gaussian elimination algorithm would require memory of size $2|S|$. Instead, the matrix can also be inverted more memory-efficiently by computing the inverse via the

LU decomposition and doing most of the computatons in-place. The required memory to obtain S from S^{-1} with this approach amounts to $|S| + 2(n-k)$ bytes where $2(n-k)$ bytes are used to temporarily store the permutation matrix that is created by the LU decomposition.

Table 3.8 depicts the memory that is required to compute the smaller matrix S instead of computing the public key T by using the Gaussian elimination algorithm.

Parameter Set	Compute T	Compute S
mceliece348864	334,848	75,264
mceliece460896	718,848	197,184
mceliece6688128	1,391,104	349,440
mceliece8192128	1,703,936	349,440

Table 3.8: Comparison of the memory footprint (in bytes) for producing the public key T or the matrix S, respectively. The original Classic McEliece algorithm applies Gaussian elimination to the $(n-k) \times n$ matrix $\hat{H}$ to produce T. Computing S instead of T to facilitate streaming the public key results in a significantly lower memory requirement.

Table 3.9 compares the size of the Classic McEliece key pair with the size of only the private key combined with the matrix S that is computed instead of the public key.

Parameter Set	Key Pair Size	Private Key + S
mceliece348864	267,572	81,268
mceliece460896	537,728	204,672
mceliece6688128	1,058,884	360,580
mceliece8192128	1,371,904	363,776

Table 3.9: Size of the Classic McEliece key pair compared to the size of the private key and the matrix S in bytes.

In practice, the public key size dominates the memory requirements of the Classic McEliece scheme, thus the replacement of the public key by the matrix S is a significant reduction for the memory requirements. Since it is an unusual concept, it is explained again without the technical details: On the side of the private key holder, the matrix S is computed instead of the public key. On the receiving end, i.e., another party that wants to perform the encapsulation operation, of course the public key is required. The matrix S shall never be made public. By utilizing the matrix S, the public key can be streamed from the private key holder, by utilizing the relationship $H = (I \mid T) = S\hat{H}$. It is easy to memory-efficiently compute rows or columns of the public key T from this. In summary, this reduces the memory requirements to only handle the matrix S instead of the whole public key (or even the somewhat larger matrix H). As an example, no Classic McEliece key pair could be computed on a device with 256 kilobytes of RAM. However, computing the private key and S, and subsequently streaming the public key, is possible for both the mceliece348864 and the mceliece460896 parameter sets.

Due to the modified encapsulation that enables streaming, the encapsulation operation can be performed on almost any embedded device since it only requires up to 3 kilobytes. This enables the use even for low-memory smartcards.

Streaming of SPHINCS$^+$ Signatures

SPHINCS$^+$ is a hash-based signature scheme and employs a hypertree structure that is comparable to XMSS. The hypertree consists of multiple layers of Merkle signature trees. These trees have WOTS$^+$ one-time signatures at their leaves. A notable difference to XMSS is, that the bottom layer of the SPHINCS$^+$ hypertree employs FORS few-time signatures at the leaves. The FORS signatures directly sign message digests, whereas the one-time signatures in the inner Merkle trees are used to sign the next layer in the hypertree.

The reader is referred to the round-3 supporting documentation of SPHINCS$^+$ for more details on the scheme[9] and some familiarity with SPHINCS$^+$ is assumed.

Parameter Set	n	h	d	$\log(t)$	k	w	Signature Size
SPHINCS$^+$-128f	16	66	22	6	33	16	17088
SPHINCS$^+$-128s	16	63	7	12	14	16	7856
SPHINCS$^+$-192f	24	66	22	8	33	16	35664
SPHINCS$^+$-192s	24	63	7	14	17	16	16224
SPHINCS$^+$-256f	32	68	17	9	35	16	49856
SPHINCS$^+$-256s	32	64	8	14	22	16	29792

Table 3.10: SPHINCS$^+$ parameter sets as proposed for NIST Round 3. The signature size is given in bytes. Further, the parameters are listed, n is the security parameter (in bytes), h is the height of the hypertree, d is the number of the layers in the hypertree, $\log(t)$ is the height of the FORS trees, k is the number of the FORS trees, and w is the Winternitz parameter.

As can be seen in Table 3.10, the size of SPHINCS$^+$ signatures is considerable with up to almost 50 kilobytes. In the following, it is elaborated on how the signature can be streamed.

A SPHINCS$^+$ signature consists of multiple parts as illustrated in Figure 3.1. First, there is the randomness R followed by a FORS signature. The FORS signature itself consists of k private key values, each combined with an authentication path in the corresponding Merkle tree. A hypertree signature follows which consists of d Merkle signatures, which each again consists of a WOTS$^+$ signature and an authentication path. A WOTS$^+$ signature is the concatenation of multiple hash-chain nodes that are computed by the WOTS$^+$ chaining function.

Each of the described parts, in turn, consists of one or more n-byte blocks. More precisely, the randomness R, the FORS private key values, every node in the authentication paths, as well as every hash-chain node are all separate byte arrays of size n — because

[9] `https://sphincs.org/data/sphincs+-round3-specification.pdf`

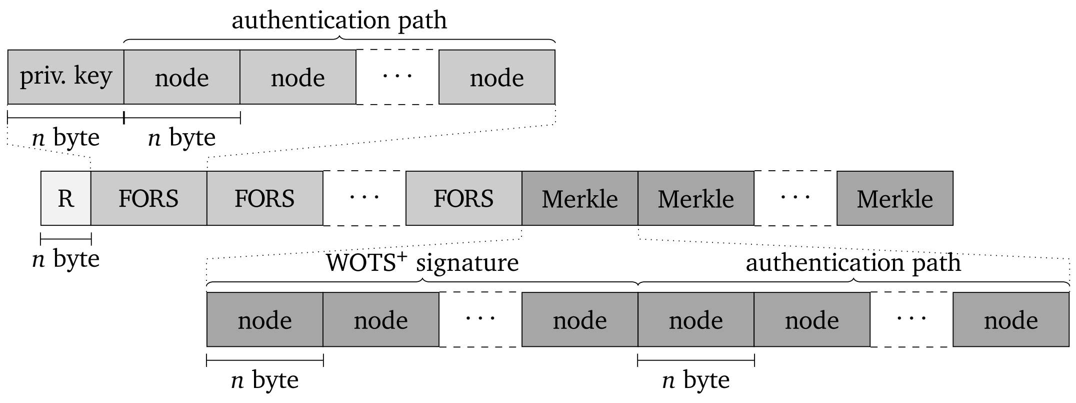

Figure 3.1: SPHINCS$^+$ signature format.

they are each the result of an invocation of the instantiated hash functions that output n bytes. This makes it natural to split signatures into chunks of n bytes as well.

The chosen approach to enable efficient streaming of SPHINCS$^+$ is to replace the signature buffer with the abstract model of a buffer which is described earlier in this section. That is, the signature buffer can be replaced by storing the currently relevant n-byte chunk of the signature which is accessed in sequential order.

In the following, this is described in some more detail from a signer's perspective. Although the computations are slightly different for the verifier, it is easy to see that the same general abstraction applies. The components of a SPHINCS$^+$ signature are:

- R: This value is generated by combining the message with a secret PRF key and optional randomness using a hash function. After it has been used to compute the message digest, which is going to be signed by FORS, the value can be written to the stream and removed from memory.
- FORS: A FORS signature consists of private keys that are released as part of the signature and an authentication path in a corresponding Merkle tree:
    - Private Keys: A FORS private key is computed by applying the PRF to the SPHINCS$^+$ secret key seed and the appropriate FORS address. It can be immediately streamed out.
    - Authentication Path: The authentication-path node at a given height is the root of a subtree and can be computed with the tree-hash algorithm directly, given that the index of the left-most leaf in this subtree is specified. In this way it is possible to compute the nodes of the authentication path in order and the buffer for the authentication path can be omitted. To avoid costly recomputations of leaf nodes, the root-node computation is intertwined with the computation of the authentication-path nodes.
    - Public Key: FORS public keys are only an implicit part of the signature, but handling them in an efficient streaming implementation warrants some attention to avoid recomputations of nodes and memory overhead. The

FORS root nodes are combined to form the public key by computing a tweakable hash over the nodes. This value is then signed by the bottom layer of the SPHINCS$^+$ hypertree. Instead of buffering all root nodes in order to compute the tweakable hash function, a state of the tweakable hash function is maintained and updated with each newly computed root node, after streaming the corresponding authentication path.

- SPHINCS$^+$ Hypertree: The signature components of the SPHINCS$^+$ hypertree consist of WOTS$^+$ signatures and nodes in the authentication paths:
  - WOTS$^+$ Signature: A WOTS$^+$ signature consists of a number of hash-chain nodes that are derived from the private key values and authenticates the tree on the layer below. The computation is typically intertwined with the computation of the WOTS$^+$ public key that is also needed for generating the corresponding leaf in the hypertree. In order to compute the public key, all hash-chain nodes are computed in order. Writing out the signature thus basically only means writing out the correct node while the public key is computed. Finally, for computing the corresponding leaf in the hypertree, the public key values are combined with a tweakable hash-function call. Instead of buffering the end node of each hash chain and then applying the tweakable hash function, the tweakable hash function can be updated after each hash-chain end node has been computed.
  - Authentication Path: This is analogous to FORS authentication paths.

Since all elements of the signature can be computed in order, the signature buffer can be reduced to just processing the current chunk of n bytes.

By applying these changes throughout the reference code, the total (peak) memory usage does not exceed 3 kilobytes for any parameter set for the three operations key generation, sign, verify, as can be seen in Table 3.11. In contrast, the reference and pqm4 implementation each additionally require memory for the signature buffer (depicted as $+sig$). The streaming implementation can freely choose a buffer size for the signature (at least n bytes, depicted as $+buf$).

The streaming approach has been integrated into a proof-of-concept TPM 2.0 implementation to show the feasibility of the approach and evaluate it in the presence of I/O overhead. Use cases like secure boot and secure firmware updates can be made feasible with a SPHINCS$^+$ streaming implementation since the originally prohibitive memory requirements due to the large signatures are lifted. The verification operation is (compared to signature and key generation) reasonably efficient on embedded devices if there are no strict requirements for the verification time. There are no significant slowdowns in the streaming implementation aside from possibly increased I/O which is outside the scope of the streaming SPHINCS$^+$ code.

Parameter Set	Sig.	Key Generation			Signing			Verification		
		ref.	pqm4	strm.	ref.	pqm4	strm.	ref.	pqm4	strm.
sphincs-sha256-128f-robust	17,088	3688	2256	1960	3176 + sig.	2320 + sig.	2000 + buf.	3344 + sig.	2808 + sig.	1728 + buf.
sphincs-sha256-128s-robust	7856	3984	2472	2056	3264 + sig.	2544 + sig.	2088 + buf.	2592 + sig.	2112 + sig.	1656 + buf.
sphincs-sha256-192f-robust	35,664	6536	3680	2192	5336 + sig.	3832 + sig.	2264 + buf.	4848 + sig.	4040 + sig.	1856 + buf.
sphincs-sha256-192s-robust	16,224	6928	4104	2336	5472 + sig.	3992 + sig.	2360 + buf.	4728 + sig.	3376 + sig.	1888 + buf.
sphincs-sha256-256f-robust	49,856	10456	5792	2456	8272 + sig.	5760 + sig.	2512 + buf.	7424 + sig.	5656 + sig.	2088 + buf.
sphincs-sha256-256s-robust	29,792	10816	6064	2584	8400 + sig.	5904 + sig.	2616 + buf.	7424 + sig.	5360 + sig.	1960 + buf.
sphincs-sha256-128f-simple	17,088	2904	2104	1632	2392 + sig.	2168 + sig.	1688 + buf.	2592 + sig.	2656 + sig.	1384 + buf.
sphincs-sha256-128s-simple	7856	3096	2432	1736	2480 + sig.	2392 + sig.	1768 + buf.	1904 + sig.	1960 + sig.	1296 + buf.
sphincs-sha256-192f-simple	35,664	5072	3520	1840	3872 + sig.	3560 + sig.	1896 + buf.	3816 + sig.	3880 + sig.	1480 + buf.
sphincs-sha256-192s-simple	16,224	5464	3944	1992	4008 + sig.	3832 + sig.	2016 + buf.	3160 + sig.	3216 + sig.	1456 + buf.
sphincs-sha256-256f-simple	49,856	8160	5512	2080	5872 + sig.	5592 + sig.	2104 + buf.	5424 + sig.	5488 + sig.	1876 + buf.
sphincs-sha256-256s-simple	29,792	8416	5896	2216	6000 + sig.	5736 + sig.	2232 + buf.	5024 + sig.	5080 + sig.	1868 + buf.
sphincs-shake256-128f-robust	17,088	4052	2012	2336	3544 + sig.	2176 + sig.	2392 + buf.	3708 + sig.	2556 + sig.	2208 + buf.
sphincs-shake256-128s-robust	7856	4352	2336	2432	3632 + sig.	2288 + sig.	2464 + buf.	2956 + sig.	1860 + sig.	2120 + buf.
sphincs-shake256-192f-robust	35,664	6892	3436	2560	5696 + sig.	3576 + sig.	2632 + buf.	5204 + sig.	3788 + sig.	2328 + buf.
sphincs-shake256-192s-robust	16,224	7288	3856	2704	5832 + sig.	3736 + sig.	2728 + buf.	4980 + sig.	3124 + sig.	2176 + buf.
sphincs-shake256-256f-robust	49,856	10912	5436	2816	8624 + sig.	5504 + sig.	2872 + buf.	7880 + sig.	5404 + sig.	2448 + buf.
sphincs-shake256-256s-robust	29,792	11168	5816	2944	8752 + sig.	5648 + sig.	2976 + buf.	7772 + sig.	4996 + sig.	2320 + buf.
sphincs-shake256-128f-simple	17,088	3476	2012	2104	2968 + sig.	2068 + sig.	2144 + buf.	3164 + sig.	2556 + sig.	1960 + buf.
sphincs-shake256-128s-simple	7856	3776	2336	2200	3056 + sig.	2288 + sig.	2232 + buf.	2468 + sig.	1860 + sig.	1800 + buf.
sphincs-shake256-192f-simple	35,664	5660	3436	2320	4464 + sig.	3468 + sig.	2368 + buf.	4404 + sig.	3788 + sig.	1956 + buf.
sphincs-shake256-192s-simple	16,224	6056	3856	2464	4600 + sig.	3736 + sig.	2488 + buf.	3748 + sig.	3124 + sig.	1936 + buf.
sphincs-shake256-256f-simple	49,856	8644	5436	2568	6464 + sig.	5504 + sig.	2592 + buf.	6012 + sig.	5404 + sig.	2072 + buf.
sphincs-shake256-256s-simple	29,792	9008	5816	2696	6592 + sig.	5648 + sig.	2712 + buf.	5612 + sig.	4996 + sig.	2112 + buf.

Table 3.11: Stack sizes for the sphincs-sha256-* and sphincs-shake256-* parameter sets for the SPHINCS$^+$ reference implementation (“ref”), pqm4/PQClean (“pqm4”), and our streaming interface implementation (“strm.”) as well as the corresponding signature sizes in bytes. The buffer size for our streaming implementation can be freely chosen to a value of at least n (hash function output size) or greater to satisfy the available memory resources.

3.3.2 Carry-Less to BIKE faster

The Key Encapsulation Mechanism BIKE has been selected as candidate for potential standardization after round 3 of the NIST PQC competition. We have optimized BIKE for the ARM Cortex-M4 and the RISC-V-based VexRiscv platform and achieved new speed records for embedded constant time implementations. Therefore, we leverage the performance benefit of the bit-polynomial multiplication in radix-16 representation. The full paper is available at [Che+22].

Bit-Polynomial Multiplication via Integer Multiplication.

For the acceleration, we chose an uncommon option to implement bit-polynomial multiplication that uses integer multiplication in combination with data in a radix-16 representation. With the radix-16 representation, one expresses a degree-7 polynomial $a = \sum_{i=0}^{7} a_i x^i \in \mathbb{F}_2[x]$ as a 32-bit integer $a_0 + a_1 2^4 + a_2 2^8 + \cdots + a_7 2^{28}$. Multiplying polynomials $a \cdot b \to c$ in this form with integer multiplication yields:

$$\begin{aligned}(a_0 + a_1 \cdot 2^4 + a_2 \cdot 2^8 + \cdots + a_7 \cdot 2^{28}) \cdot (b_0 + b_1 \cdot 2^4 + b_2 \cdot 2^8 + \cdots + b_7 \cdot 2^{28})\\ = a_0 b_0 + (a_1 b_0 + a_0 b_1) \cdot 2^4 + (a_2 b_0 + a_1 b_1 + a_0 b_2) \cdot 2^8 + \cdots + (a_7 b_7) \cdot 2^{56}\end{aligned}$$

an integer where the bit of index $4i$ is exactly c_i, and thus after masking out the other indices remains c in radix-16 representation. Chen and Chou [CC21b] presented the multiplication in radix-16 formats and applied it to multiplication in $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$, i.e., polynomials of 12 and 13 bits. We present the techniques for extending the method to bit-polynomial multiplication in BIKE, including the optimization of 32-bit base multiplication, data conversion, logic shift operation, and building multiplications for polynomials of various sizes in the following.

Base Multiplication in Radix-16 Form.

```
     00010001000100010001000100010001 × 00010001000100010001000100010001
-------------------------------------------------------------------------------
                                        0001 0001 0001 0001 0001 0001 0001 0001
                                   0001 0001 0001 0001 0001 0001 0001 0001
                              0001 0001 0001 0001 0001 0001 0001 0001
                         0001 0001 0001 0001 0001 0001 0001 0001
                    0001 0001 0001 0001 0001 0001 0001 0001
               0001 0001 0001 0001 0001 0001 0001 0001
          0001 0001 0001 0001 0001 0001 0001 0001
     0001 0001 0001 0001 0001 0001 0001 0001
-------------------------------------------------------------------------------
0000 0001 0010 0011 0100 0101 0110 0111 1000 0111 0110 0101 0100 0011 0010 0001
```

Figure 3.2: Performing an 8-bit bit-polynomial multiplication with 32-bit integer multiplication in radix-16 form.

Figure 3.2 shows an example of an 8-bit bit-polynomial multiplication with a 32-bit integer multiplication using the radix-16 form. Even during a multiplication of two radix-16 values with all bits set, the carry bits do not propagate. Furthermore, this example demonstrates that the lower half of the multiplication result can be added to the higher half (both residing in a 32-bit register) without a prior reduction. One can see that the pairwise sum of each nibble is capped at eight and therefore does not lead to a carry propagation to the next nibble. This observation allows the use of the powerful multiply-with-accumulate instruction (`umlal`) of the Cortex-M4 during the combination of 16 8-bit bit-polynomial multiplications to perform one 32-bit multiplication.

The radix-16 format quadruples the size of a polynomial during computation, to avoid this memory overhead when storing polynomials, we pack four bytes in radix-16 format in one register by shifting byte i, i bits to the left.

To perform one 32-bit bit-polynomial multiplication we therefore extract four bytes for each operand, perform 16 integer multiplications with reductions in between and pack the result in two 32-bit registers. For the Cortex-M4 we are able to express this operation with only 46 instructions, by using the `umlal` instruction as explained before and the barrel-shifter. For our RISC-V implementation we need 89 instructions for the same operation, because not only of the missing barrel-shifter and multiply-with-accumulate instruction, but also are multiplications of 32-bit values expressed with two instructions, one for the lower half and one for the upper half of the result.

Performance of BIKE KEM

Measuring the three KEM operations for BIKE demonstrates that our radix-16 multiplication approach beats the current fastest implementations of BIKE on the Cortex-M4. For the `bikel1` parameter set we observe an improvement in cycles of about 13% for the key generation and about 7% for the encapsulation. The decapsulation is about 3% slower than the FFT based approach. This is due to the many multiplications in the decoder where input transformations can be omitted and thus the FFT based multiplications are faster. For an implementation where code size is insignificant, one could use the FFT based multiplication for the decoder and our radix-16 multiplication in the key generation and encapsulation to achieve the best overall performance. The results are shown in Table 3.12.

Table 3.12: Cycle counts for BIKE on the Cortex-M4.

	Key Gen	Encaps	Decaps	Impl
	21,137,291	2,989,187	50,832,769	Radix-16
`bikel1`	24,935,033	3,253,379	49,911,673	FFT [CCK21]
	65,414,337	4,824,059	114,592,442	`portable` [Ara+17]

3.3.3 Optimizing Kyber and NewHope Ciphers on RISC-V

This section covers the implementation of the polynomial arithmetic used by NewHope and Kyber, using the standardized RISC-V ISA. This work was done as part of [Alk+20]. In Section 3.3.4, we describe the implementation of the finite field arithmetic used

Algorithm 1: Barrett reduction on 32-bit inputs.

```
Input: Integer $a < 2^{32}$              (register a0)
Input: Prime $q$                          (register a1)
Input: Integer $\lfloor \frac{2^{32}}{q} \rfloor$            (register a2)
Output: reduced $a < 2^{14}$             (register a0)
mulh t0, a0, a2 // $t \leftarrow a \cdot \lfloor \frac{2^{32}}{q} \rfloor$;
mul t0, t0, a1 // $t \leftarrow t \cdot q$;
sub a0, a0, t0 // $a \leftarrow a - q$;
```

Algorithm 2: 4× interleaved Barrett reduction on 32-bit inputs.

```
Input: Integer $a_i < 2^{32}$, with $1 \le i \le 4$
                              (registers a0 to a3)
Input: Prime $q$                          (register a4)
Input: Integer $\lfloor \frac{2^{32}}{q} \rfloor$            (register a5)
Output: reduced $a < 2^{14}$             (register a0)
mulh t0, a0, a5 // first batch;
mulh t1, a1, a5 // second batch;
mulh t2, a2, a5 // third batch;
mulh t3, a3, a5 // forth batch;
mul t0, t0, a4 // first batch;
mul t1, t1, a4 // second batch;
...;
sub a0, a0, t0 // first batch;
...
```

for the coefficients of the polynomials. Section 3.3.5 follows with details on the NTT implementation and related operations on polynomials.

3.3.4 Finite Field Arithmetic in Kyber and NewHope

The reference implementations of both Kyber and NewHope, use two reduction algorithms for multiplication and addition/subtraction to be able to make use of specific 16-bit operations. The RISC-V ISA, however, does not support 16-bit arithmetic operations. The bulk of the arithmetic in Kyber and NewHope concerns polynomials that have 256, 512, or 1024 coefficients, each of which being an element in $\mathbb{Z}_q$ with either a 14-bit ($q = 12289$) or a 12-bit ($q = 3329$) prime. To avoid the need for handling underflows during the Montgomery reduction step, the reference implementations use a signed 16-bit integer representation. This approach does not lend itself well to the RISC-V ISA, as the RISC-V does not feature a sign-extension operation.

A 32-bit reduction algorithm is the better choice for the RISC-V platform, as it features an instruction that returns the high 32-bit of a 32×32-bit multiplication. This can effectively be used to to remove shifting operations during the modular reduction. This approach lends itself well to a 32-bit Barrett reduction for a better fit to the target ISA, as shown in Algorithm 1, requiring only three instructions. Furthermore, unlike the Montgomery reduction, this avoids the need for any base transformation of the coefficients.

The instructions of the reduction may block deeper processor pipelines considerably, as each instruction depends on the result of its predecessor (see Algorithm 1). This usually blocks the next instruction until the current instruction is retired, unless the processor supports some form of pipeline bypass. The RISC-V ISA, however, features a large number of registers, which allows for some instruction interleaving and considerably alleviating the problem. Algorithm 2 shows in abbreviated form the interleaving of the execution of four independent reductions to avoid pipeline stalls. This optimization is

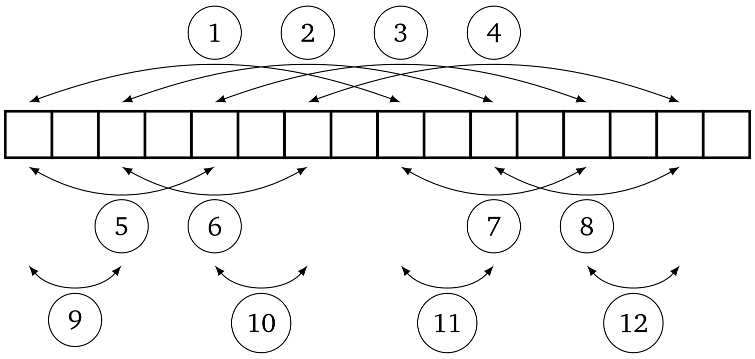

Figure 3.3: The order of butterfly operations for an NTT with three merged layers.

not relevant for RISC-V CPUs with shorter pipelines, but does not harm the performance, as the number of instructions remains the same as with four consecutive reductions. The same optimization can also be applied to the polynomial multiplication, addition, and subtraction.

3.3.5 Number Theoretic Transform

Both the NewHope and Kyber ciphers make heavy use of the NTT, which is used to speed up polynomial multiplication. A common measure to reduce the number of memory loads and stores in software implementations is the merging of several layers of the NTT. The NTT transformation requires powers of the primitive n-th root of unity during butterfly operations, which are called twiddle factors, which are usually precomputed and stored in memory. In unmerged implementations one pair of coefficients is loaded from memory, the appropriate twiddle factor from memory is loaded, the butterfly operation is performed, and the result is stored. In merged implementations, as illustrated in Figure 3.3, the number of loaded pairs and butterfly operations per merged layer is doubled, i.e., for l layers, 2^l coefficients and $2^l - 1$ twiddle factors are loaded and $2^l \cdot l$ butterfly operations can be performed before storing the results. This optimization lends itself well to the RISC-V ISA as it features a sufficient amount of registers to merge up to three layers, i.e., eight coefficients are processed with twelve butterfly operations before the next set is loaded. Four butterfly operations can further be interleaved to further avoid pipeline stalls as described above.

A moderate amount of code unrolling can be employed, e.g., to enable interleaving. But due to the high number of registers of the RISC-V platform, unrolling is not necessary to free registers otherwise used for counters. Accordingly, using a looped approach for the iteration through the merged layers can still be used to reduce the code size.

3.4 Design and Implementation of PQC-Based Protocols

Existing cryptographic protocols are built with classic algorithms like RSA and elliptic-curve-based cryptosystems in mind. In principle PQC algorithms can replace those by replacing classic signature schemes with PQC schemes, and by replacing classic public-key-encryption and key-exchange schemes with PQC KEMs. In practice there are some challenges and considerations. In the following, this is evaluated for the widely deployed X.509 standard and the TLS protocol. The integration of PQC into X.509 certificates plays a crucial role for the Public Key Infrastructure (PKI)-based authentication in many cryptographic protocols like TLS. Different approaches and considerations are presented.

First, some details of the X.509 standard and then the TLS 1.3 protocol are presented as far as it is necessary for this report. Then, different aspects of PQC integration are covered and the current state in research and in open source libraries is explored. Since TLS uses the X.509 standard in its handshake protocol, some aspects are overlapping between the two standards.

3.4.1 X.509 Certificates

RFC 5280[10] describes X.509 v3 certificates and X.509 v2 certificate revocation lists for use in the internet in the context of PKIs. The primary use of X.509 certificates is to authenticate an entity by creating a binding between the provided public key and the subject of the certificate.

The ASN.1 syntax is used to describe the format of a certificate:

```
Certificate  ::=  SEQUENCE  {
        tbsCertificate       TBSCertificate,
        signatureAlgorithm   AlgorithmIdentifier,
        signatureValue       BIT STRING  }

   TBSCertificate  ::=  SEQUENCE  {
        version         [0]  EXPLICIT Version DEFAULT v1,
        serialNumber         CertificateSerialNumber,
        signature            AlgorithmIdentifier,
        issuer               Name,
        validity             Validity,
        subject              Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                             -- If present, version MUST be v3
        }
```

A certificate consists of a *SEQUENCE* of the three fields *tbsCertificate, signatureAlgorithm*, and *signatureValue* where the *signatureValue* field contains the signature that authenticates the certificate.

The *SubjectPublicKeyInfo* field and its subfields are of particular interest.

```
   SubjectPublicKeyInfo  ::=  SEQUENCE  {
        algorithm            AlgorithmIdentifier,
        subjectPublicKey     BIT STRING  }
```

[10]https://datatracker.ietf.org/doc/html/rfc5280

```
AlgorithmIdentifier  ::=  SEQUENCE  {
     algorithm               OBJECT IDENTIFIER,
     parameters              ANY DEFINED BY algorithm OPTIONAL  }
```

Here the *algorithm* field contains an Object Identifier (OID) that uniquely identifies the used public-key algorithm. The *subjectPublicKey* field contains the public key itself as a *BIT STRING* field which can in itself contain an ASN.1 structure.

3.4.2 TLS 1.3

The Transport Layer Security Protocol (TLS) is used to establish secure connections over TCP/IP. Connections made with TLS 1.3 provide confidentiality, authenticity, and integrity. TLS is divided into the record protocol and the handshake protocol. The handshake protocol is responsible for the setup of the TLS connection, i.e., authenticating the server and optionally the client, as well as exchanging encryption keys.

Public keys for the key exchange are transmitted in the *Key Share Extension*, an extension that can be sent in the Client Hello and the Server Hello message. It is used to convey the (EC)DHE key share of the respective party. The key share consists of a named group and the (EC)DHE public key. A key share can at most be $2^{16} - 1$ bytes long. The key shares are subsequently used to derive a shared secret and use it in the TLS 1.3 key schedule.

With the use of signature algorithms, a server – and optionally the client – authenticate themselves. The *Certificate Verify* message follows after the *Certificate* message and proves the possession of the private key that corresponds to the public key in the certificate by including a signature over the hash of the transcript of sent handshake messages. A signature in this message can at most be $2^{16} - 1$ bytes long. For the corresponding public key in the certificate, there is no such limitation, however, the length of a certificate chain is limited to $2^{24} - 1$ bytes in the TLS 1.3 handshake.

3.4.3 Existing PQC Protocol Approaches and Implementations

There are many different drafts and proposals to adapt the TLS and X.509 protocol to incorporate new PQC schemes. A selection of relevant approaches is presented here.

TLS

While there is substantial work for TLS 1.2 as well, the focus in the following is on TLS 1.3. The new TLS 1.3 version is expected to be more relevant in the future, even though, TLS 1.2 is still used, and even though there are also proposals to change TLS 1.2, e.g. [CC21a].

Quantum Safe Cryptography Key Information [Vre+22] This draft addresses the binary representation of PQC keys. Without a defined format for keys, their serialization and encoding formats, there can be no interoperability. The draft further addresses key compression formats. In order to make the keys addressable, OIDs are assigned for key algorithms as well as key parameters.

A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret [SS17] The idea in this draft is to add an additional key share extension, named *additional_key_share*. The extension can be optionally sent and contains an up to $2^{16} - 1$ byte public key or ciphertext. The key schedule is additionally extended by HKDF-Extract and HKDF-Expand calls to incorporate the additional shared secret.

Hybrid key exchange in TLS 1.3 [SFG22] This draft aims to add hybrid key exchange methods while staying close to the TLS 1.3 key exchange to avoid changing existing data structures. The public keys and ciphertexts are respresented as a *KeyShareEntry* which is up to $2^{16} - 1$ bytes. Combining the schemes is done by concatenation, i.e., two shared secrets are simply concatenated and then inserted into the TLS 1.3 key schedule (in place of the (EC)DHE shared secret). Each new hybrid key exchange combination is viewed as a new key exchange method from the protocol's view. Design goals of the approach are backwards compatibility, high performance, low latency, no extra round trips, and minimal duplicate information. However, the proposal has the drawback of duplicate shares if an algorithm is offered in two different combinations. An experimental implementation of this approach is available as an OpenSSL fork by the Open Quantum Safe project[11] [Ste+].

Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3 [Why+17] This is another draft that aims to include hybrid key-exchange schemes into TLS 1.3. The newly proposed key exchange algorithms are treated as (EC)DH groups as much as possible, i.e., a new PQC KEM is added as a group. The keys and ciphertext are again conveyed via the *KeyShareEntry* struct that is already defined in TLS 1.3 and as for some of the other presented drafts, the individual keys are concatenated and used instead of the (EC)DHE shared secret. Some additional logic is added in order to negotiate the hybrid key exchange and no restrictions are made on which algorithms or how many algorithms are combined for the hybrid key exchange. Further, the draft takes care to only include key-share material once, even if it appears in multiple hybrid key-exchange combinations.

Hybrid ECDHE-SIDH Key Exchange for TLS [KK18] This is a specific proposal to integrate two specific key exchange combinations: *X25519* with *SIDH503* and *X448* with *SIDH751*. As for the *Hybrid key exchange in TLS 1.3* [SFG22], the shared secret is derived by concatenating the individual shared secrets of the constituent algorithms and using it in place of the (EC)DHE shared secret. The draft defines new groups for the proposed key exchange combinations, as well as a format for the SIDH key exchange.

KEM-based Authentication for TLS 1.3 [Cel+22] The KEM-based Authentication for TLS 1.3 draft is based on the research on KEMTLS, a TLS variant without handshake signatures [SSW20], [SSW21]. The basic idea is that server and client certificates make use of KEMs instead of signatures, i.e., static long-term KEM keys are used instead of signature keys. This is in some aspects similar to the old (TLS 1.2 and below) RSA key

[11]`https://github.com/open-quantum-safe/openssl`

exchange. When exchanging a secret by encapsulating with the peer's long-term public key, the peer is implicitly authenticated by being able to decapsulate the ciphertext. By using an additional ephemeral KEM key in the handshake, forward-secrecy can be achieved. KEMTLS also describes an abbreviated handshake when the server's public key is already known. In this case the client can directly send the encapsulation to the server in a ClientHello extension.

X.509

In the following, relevant approaches to integrating PQC into X.509 are presented.

Algorithms and Identifiers for Post-Quantum Algorithms [Mas+22] The draft aims to specify post-quantum signature algorithms for the use in the internet X.509 public key infrastructure. It will reference NIST standards, once finished, and also use the OIDs that will be specified by NIST. The draft will specify ASN.1 public and private key encodings, as well as the ASN.1 encoding of the signature value.

Algorithm Identifiers for NIST's PQC Algorithms for Use in the Internet X.509 Public Key Infrastructure [Tur+22] Analogously to [Mas+22], this draft aims to specify post-quantum KEM algorithms for the use in the internet X.509 public key infrastructure. Likewise, it will refer to the NIST standard and provide the respective encodings for the ASN.1 public and private key.

Composite Schemes For Use In Internet PKI This paragraph handles the related drafts *Composite KEM For Use In Internet PKI* [OG22], *Composite Encryption For Use In Internet PKI* [OGM22], *Explicit Pairwise Composite Keys For Use In Internet PKI* [OPK22], and *Composite Signatures For Use In Internet PKI* [OP22]. All of these drafts have in common, that a composite approach for combining multiple algorithms is followed. Composite means, that the structures for the signatures and ciphertexts and public and private keys appear as a single entity from the protocol's view, but the internal structure is composed of multiple different algorithms. In [OPK22], each combination of different keys is assigned its own OID, and ASN.1 structures for the combinations are defined. Likewise, this is done for the other aspects of composite post-quantum algorithms in [OG22], [OGM22], [OP22]. The combinations themselves are generic in the way, that different algorithms are combined in identical ways. A specific set of algorithms may be identified by an OID (called *explicit variants* in the drafts), but may also be generically combined by specifying the algorithms and parameters. In the new draft *Explicit Pairwise Composite Keys For Use In Internet PKI* [OMG22], that aims to replace [OPK22], feedback from the LAMPS IETF working group that might want to adopt the draft is addressed, where no generic algorithm combinations are allowed, but only the ones that can be identified by OIDs.

Non-Composite Schemes For Use In Internet PKI The draft *Non-Composite Hybrid Authentication in PKIX and Applications to Internet Protocols* [BGJ22a] proposes a non-composite approach to combine multiple algorithms and gives an overview of hybrid (multi-algorithm) authentication. The draft *Related Certificates for Use in Multiple Authentications within a Protocol* [BGJ22b] in contrast makes an explicit proposal for a new CSR attribute, bindingRequest, and a new X.509 certificate extension. With the *BoundCertificates* extension, multiple certificates are bound together. As an example, when issuing a PQC certificate, the extension can be used, to provide additional information to tie the PQC certificate to a classical certificate. Issues regarding the two prominent protocols that make use of certificate-based authentication, TLS 1.3 and IKEv2, are also addressed in the draft.

Other

It shall be briefly noted that there are many projects that cover other common protocols as well that have not been a focus in this project. There is substantial work for the VPN protocol[12,13,14,15], the SSH protocol[16] [CPS19a], and others like PQC approaches to Blockchains [Das+20].

3.4.4 Challenges regarding the Integration of PQC in TLS and X.509

For X.509 there exist no principle obstacles to including new PQC schemes from a technical point of view. The ASN.1 syntax is quite flexible and allows for arbitrary-sized keys, ciphertexts, and signatures. However, a consensus has to be reached regarding the transition from the classical era to the post-quantum era. One widely discussed approach is to make use of hybrid algorithms where the challenge is to find consensus on how to combine them (e.g. composite or non-composite and through what specific mechanism). The deployment of PQC or classic-PQC-hybrid certificates can only start when standards are finished and OIDs are assigned to algorithms.

For TLS 1.3 there exist some more challenges from a technical point of view. Generally, the TLS 1.3 standard is flexible through the use of defining new extensions. However, it has not been written with post-quantum cryptography in mind. Instead, the key exchange is tailored to the (EC)DHE key exchange variants, as can be seen in the naming of the respective *key share* extension. While signature algorithms in the post-quantum world behave like signatures in the pre-quantum world, NIST will standardize KEMs which is a different kind of algorithm than a key agreement algorithm like (EC)DHE. Further, the increased sizes of PQC algorithms have not been taken into account.

Public Key Sizes in TLS 1.3. Key shares in TLS 1.3 only allow up to $65{,}535$ bytes. This makes it impossible to directly include algorithms like Classic McEliece with considerably larger keys [CPS19b].

[12] `https://www.forschung-it-sicherheit-kommunikationssysteme.de/projekte/quasimodo`
[13] `https://pq-vpn.de/`
[14] `https://datatracker.ietf.org/doc/draft-ietf-ipsecme-ikev2-multiple-ke/`
[15] `https://datatracker.ietf.org/doc/draft-ietf-ipsecme-ikev2-intermediate/`
[16] `https://datatracker.ietf.org/doc/html/draft-kampanakis-curdle-pq-ssh-00`

Signature Sizes in TLS 1.3. For X.509 certificate chains there is a length limitation of $2^{24} - 1$ bytes, however, this is not a practical limitation and is large enough to contain many certificates with signatures of any of the round 2 submissions. In contrast, signature sizes in the TLS 1.3 *CertificateVerify* message are limited to $2^{16} - 1$ bytes. Some signature schemes, for example Picnic-{L3,L5}-{FS,UR}, have signature sizes larger than that.

Implementation-related Issues for TLS 1.3 libraries. In [CPS19b] it is also explored what the practical limits in OpenSSL 1.1.1's TLS 1.3 implementation are. That means, while the specifications define a maximum size for certain fields, the OpenSSL implementation, or any other TLS implementation for that matter, can have further restrictions due to code design choices. Since classic signature and key exchange algorithms have sizes far smaller than what some PQC algorithms offer, large sizes are not as thoroughly tested and supported. As an example, the authors had to resolve an *excessive message size* error that occured when sending a ServerHello message containing a FrodoKEM-1344-{AES,SHAKE} key. The internal buffer that holds the ServerHello message is constrained to 20,000 bytes but could simply be increased to accommodate for the larger sizes. Further, the authors note that the certificate message that contains all sent X.509 certificates, is limited to 102,400 bytes. While the limit is generous for classic cryptography, some Rainbow parameter sets produce larger keys than this limit. The authors were able to raise the limit to the protocol's limit of $2^{24} - 1$ bytes to resolve the problem. Another restriction is imposed on the maximum signature size in the CertificateVerify message and only signatures up to 2^{14} bytes can be handled by OpenSSL 1.1.1. Again, raising this value to the protocol's maximum size of $2^{16} - 1$ resolves the issue. However, the authors had to increase the corresponding 2-byte-length field to 3 bytes in order to accommodate for all PQC schemes. This is a change to the TLS 1.3 specification and thus not a fix on the implementation level.

3.4.5 Classic McEliece Streaming in TLS 1.2

The proof-of-concept TLS 1.2 implementation in the paper [RKK21] has been implemented as its own cipher suite to demonstrate the real-world implications of the streaming implementation. The details of the algorithms of the streaming implementation itself can be read up upon in the paper, see [RKK21], or in the WP 2 report [MM22].

In the following the integration into TLS 1.2 is detailed. In order to increase the comprehensibility of the explanations, Figure 3.4 depicts an overview of the messages that are sent in a TLS 1.2 handshake.

To integrate Classic McEliece as a new key exchange algorithm, we defined a new cipher suite with Classic McEliece as the key exchange algorithm. The key is supposed to be ephemeral,[17] i.e., for each new connection, a new Classic McEliece key pair shall be used. A privately defined identifier for the cipher suite is used, that is not interoperable due to lack of standardization and thus omitted here. The key exchange

[17]The new algorithm supports the use of ephemeral keys, however it is up to the server to generate a new key for each connection.

```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data


   * Indicates optional or situation-dependent messages that are not
   always sent.
```

Figure 3.4: Message flow for a full handshake. Source: `https://www.rfc-editor.org/rfc/rfc5246`

is based on the RSA key exchange, and a KEM-DEM construction is used to convert the KEM to a PKE.

1. The client sends the *ClientHello* message and includes the identifier for the Classic McEliece cipher suite.
2. The server accepts the Classic McEliece cipher suite and replies with an appropriate *ServerHello* message.
3. In the *ServerKeyExchange* message, the server sends the Classic McEliece public key. In order to do so, the server first generates the extended private key. Then, the server sends chunks of single columns of the public key to the client, until all columns have been sent. The methods are described in detail in Section 3.3.1.
4. The client receives the columns of the public key and consumes each column and updates its internal state of the syndrome computation. After the last column has been received, the client possesses a shared secret and the ciphertext that corresponds to the encryption under the server's public key of that shared secret. Then the client generates a 48-byte premaster secret, analogous to the RSA key exchange, and uses the shared secret as an AES-GCM-256 key to encrypt

the premaster secret. The client sends the encrypted premaster secret in the *ClientKeyExchange* message.

5. The server is able to decrypt the encrypted shared secret and uses it as AES-GCM-256 key to decrypt the encrypted premaster secret. Now both parties can form the same master key from the premaster secret.

6. The handshake finishes and both parties can send encrypted application data.

The client, as well as the server, have successfully been run on an ARM Cortex-M4 board[18], featuring $256\,\mathrm{kB}$ RAM. For the experiments, the parameter set *mceliece348864* has been chosen which has a public key size of $261{,}120\,\mathrm{B}$. The implementation is based on the mbedTLS library[19]. The certificate chain is a full post-quantum SPHINCS$^+$-256f certificate chain. This experiment demonstrates, that post-quantum cryptography can be used on embedded devices, even if the public-key sizes seem prohibitive.

3.4.6 SPHINCS$^+$ and Streaming TPM 2.0

The streaming approach for SPHINCS$^+$ signatures in Section 3.3.1 has been integrated into the TPM 2.0 reference implementation[20] by Microsoft. Data structures for SPHINCS$^+$ signatures and commands for generating keys and signatures have been added.

In order to send commands to the TPM and to receive its responses, we implemented a variant of the TIS protocol. TIS defines a $24\,\mathrm{bit}$ address space that is mapped to control registers and data buffers inside the TPM. Read and write operations on specific addresses are used to transfer commands and data to the TPM, to receive responses from it, and to initiate the execution of commands.

The specification describes a simple SPI-based protocol for implementing read and write operations for TPMs that are not directly connected to a memory bus. Messages in this SPI protocol start with a byte that uses the first bit to indicate whether a read or write operation is performed and the remaining $7\,\mathrm{bit}$ to define the length of the data in bytes. This initial byte is followed by a $24\,\mathrm{bit}$ address indicating the source (for a read operation) or destination (for a write operation). After this, data is transferred.

The host system initiates all communication with the TPM and polls a $32\,\mathrm{bit}$ status register using the SPI-based protocol to detect if the TPM is ready to receive commands or if a response is ready. We only implemented a minimal subset of the address space that is defined in the TIS specification. The two most important addresses are the $32\,\mathrm{bit}$ status register `STS` at address `000018h` and the $32\,\mathrm{bit}$ FIFO command-and-response register `DATA_FIFO` at address `000024h`. Specific bits of the `STS` are used by the TPM to communicate its readiness to receive commands or transmit responses as well as for the host system to initiate the execution of a previously written TPM command.

We propose an extension to this communication protocol for the streaming of data between the host and the TPM. We extend the TIS interface by two addresses, the

[18]The precise model is the STM32F429ZI.

[19]`https://github.com/Mbed-TLS/mbedtls`

[20]https://github.com/microsoft/ms-tpm-20-ref/

32 bit `IOSTREAM` FIFO register at address `000030h` for the streaming of data and the 32 bit `STREAMSIZE` register at address `000040h` to communicate the size of the data to be streamed. The `STS` register is polled by the host platform regularly. Several of its bits are marked as reserved by the TIS specification. We decided to use two of those reserved bits to signal that the TPM is ready to send (`STS` bit 23) or receive (`STS` bit 24) data. If one of these bits is set when the host system reads the `STS` register, it reads out the `STREAMSIZE` register, which holds the number of bytes that the TPM is able to send or receive. The data is then read from or written to the `IOSTREAM` register, which acts as a FIFO. This helps to reduce additional polling during streaming, since the host system is already polling the `STS` register during the execution of any TPM command to detect if a response is ready.

An alternative would be to formalize the sequence of streaming messages in a state machine and to issue special commands asking for further data or to transmit data via the `DATA_FIFO` register. This approach however would introduce a significant overhead in transmitted data and all data would have to pass through the serialization and deserializing layers of the TPM firmware. Furthermore, the implementation of a cryptographic scheme using such a state-machine-based communication layer would require significant changes to the program flow into a state-machine as well instead of being able to handle I/O transparently during the cryptographic computations as described in 3.3.1 for SPHINCS$^+$.

We used the SPI controller (in the SPI "slave" role) of the STM32F4 SoC to implement the TIS protocol. Our implementation is driven by an interrupt service routine that is triggered every time a byte is received on the SPI bus. The interrupt operates a small state machine that handles writing to and reading from the available addresses, including the streaming of signature data. This approach results in an interruption of the code for every byte that is transferred. These frequent interrupts could be avoided using a dedicated hardware implementation of the TIS protocol on an actual TPM.

For our measurements with the TPM integration we used a Raspberry Pi 4 Model B in the role of the host system, because it provides an SPI controller for communication with our TPM prototype.

Table 3.13 depicts measurements that have been taken for that setup.

Parameter Set	Key Generation				Signing				Verification			
	Embedded			Host	Embedded			Host	Embedded			Host
	strm. [mcyc]	TPM [mcyc]	I/O Δ	total [s]	strm. [mcyc]	TPM [mcyc]	I/O Δ	total [s]	strm. [mcyc]	TPM [mcyc]	I/O Δ	total [s]
128f-robust	55.0	60.5	9.1 %	0.531	1280	1960	34.9 %	11.8	82.7	470	82.4 %	2.90
128s-robust	3520	3870	9.1 %	23.3	26,400	29,500	10.3 %	176	27.8	206	86.5 %	1.33
192f-robust	81.8	91.3	10.4 %	0.727	2160	3580	39.8 %	21.4	123	928	86.8 %	5.63
192s-robust	5240	5840	10.4 %	35.0	48,400	54,600	11.4 %	326	44.3	406	89.1 %	2.52
256f-robust	300	330	9.0 %	2.17	6120	8340	26.7 %	49.8	177	1320	86.6 %	7.98
256s-robust	4800	5280	9.0 %	31.6	58,900	65,800	10.5 %	392	90.0	767	88.3 %	4.67
128f-simple	27.3	29.7	8.2 %	0.340	640	1250	48.9 %	7.54	39.8	420	90.5 %	2.61
128s-simple	1750	1900	8.2 %	11.5	13,300	14,800	10.2 %	88.2	13.6	189	92.8 %	1.23
192f-simple	40.2	44.4	9.6 %	0.447	1080	2360	54.3 %	14.1	58.4	850	93.1 %	5.17
192s-simple	2570	2840	9.6 %	17.1	24,400	27,700	11.6 %	165	21.2	386	94.5 %	2.40
256f-simple	106	120	10.9 %	0.914	2230	4110	45.8 %	24.6	60.6	1180	94.9 %	7.14
256s-simple	1700	1910	10.9 %	11.6	22,000	25,800	14.7 %	154	28.9	689	95.8 %	4.21

Table 3.13: Performance data for SPHINCS$^+$ key generation, signing, and verification on the embedded device and on the host. For the embedded device, we list the cycle counts of the reference implementation including the streaming interface as “strm.” in mega cycles, the integration of SPHINCS$^+$ into the TPM prototype as “TPM” in mega cycles, and the communication overhead, i.e., the difference between the two in percent, as “I/O Δ”. For the host, we list the overall wall-clock time from issuing a TPM command until its completion (including I/O) as “total” in seconds.

3.4.7 Stateful Hash-Based Signature Schemes

This section discusses the challenges of deploying stateful hash-based signature (HBS) schemes on an abstract level. Further, an assessment is made as to whether stateful HBS are generally recommendable. The use case for a certification authority is considered.

Hash-based cryptography is an obvious candidate for post-quantum cryptography since hash functions are considered secure against quantum computer aided attacks. For example, the security of the hash-based signature schemes XMSS and SPHINCS$^+$ only rely on the properties of the underlying hash functions. In general, this leads to less assumptions than is typically the case for signature schemes: Signature schemes that are able to process arbitrary-length input, also depend on the security of the used hash functions.

Stateful HBS schemes like XMSS[21] and LMS[22], however, introduce a new problem: the state. In contrast to typical signature schemes, the private key changes after each signature to reflect the change in some internal state of the scheme. For the two mentioned schemes, the state amounts to some index to keep track of which one-time signatures have already been used. It would be fatal to reuse the same state for two signatures, as this would lead to the generation of a second one-time signature with the same one-time signature key pair. This directly reduces the security, or can completely break the scheme, such that an adversary can forge signatures.

There are a few possible pitfalls when using stateful HBS schemes. Some are easily avoided, and some are more difficult to adequately address.

1. Virtual Machines. Virtual machines offer the possibility to clone an entire (virtual) machine with its exact state. For stateful HBS schemes, this can be problematic. When cloning the VM, there exist two identical private keys with an identical state that are not necessarily aware of the second copy. It is non-trivial to assure that each state is only used once during the key lifecycle of both clones.

2. Updating Non-Volatile Memory. From a very high level perspective, memory can be divided into volatile and non-volatile memory. Volatile memory encompasses, for example, the RAM and the CPU registers, Non-volatile memory could be flash or disk storage. In contrast to volatile memory, the data in the non-volatile memory will be retained when there is no power.

   For modern systems, however, there are many in-between layers of caches, online or offline backup storage, and even software layers like file systems. Databases and caches inside of applications can constitute another layer. Each of these layers might be caching a private key without writing it to the next layer(s) in the hierarchy. As an example, modern operating systems might schedule an actual physical write operation to the non-volatile memory for another time while transparently creating the impression on the filesystem level that the write operation has already been carried out.

   Therefore, it is non-trivial to assure that the non-volatile memory is correctly updated with the new state. In normal operation, all of this is designed to be

[21] `https://datatracker.ietf.org/doc/html/rfc8391`
[22] `https://datatracker.ietf.org/doc/html/rfc8554`

transparent from a software perspective. However, events like a power shutdown can pose serious risks as they can cause inconsistencies across the different layers.

3. Backups. Backups are closely related to the previous point but deserve special attention. All the same concerns that hold for non-volatile memory in general also hold for backups.

   For stateless keys, a simple backup would suffice. For stateful keys, additonal care has to be taken, such that restoring the backup does not lead to a reused state. Therefore, either a new backup has to be created after each sign operation, or the backup recovery has to infer the state from some additional source. Further, similarly to cloning a VM, a backup may not be used to restore the same state on different machines. In any case, the state in the keys introduces new challenges for restoring backups.

4. Abortion of a Signature Generation Operation. When a signature generation operation is in progress, it has to be taken care that it is only then output, after the state is updated. If one fails to do so, a sudden abortion of the operation can lead to the private key not being updated. With the signature (or parts of it) already output, that means the state will be used twice.

5. Concurrency. In a setting where a high throughput of signatures is desired, it might be beneficial to offload the work to multiple different cores or devices.

   In the typical scenario of stateless keys, all devices could simply share the same key. For stateful schemes, additional care has to be taken, such that no state is used twice. Additional overhead for the coordination is needed.

Also see NIST's security considerations regarding stateful HBS in [ST20].

Apart from pitfalls that directly affect the security of the schemes, there are other aspects that might make stateful HBS schemes unattractive. As an example, integrating stateful HBS schemes into a crpytographic library or protocol can be challenging. Something as trivial as having a const-qualifier for the private key in the API functions can make the integration impossible without changing the API. In contrast to other signature schemes, the private key will change after the signature generation operation, thus the const-qualifier prevents the integration. Further, for stateful schemes, there is a (possibly very low) maximum number of signatures that can be created with a given key. Integrating this into the key life-cycle management might be non-trivial, and at least, poses a new special case that has to be handled.

For the reasons given in Items 1 to 5, the consortium comes to the same conclusion as NIST[23] and deems stateful HBS as not suitable for general use.

To address some of the issues described above, we implemented XMSS on a Hardware Security Module (HSM). This implementation is described in Section 4.2.2.

[23] `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-208.pdf`

3.5 Mixed PQ-PKI

In contrast to classic crpytography, PQC schemes have vastly differing characteristics and it is an interesting question, whether or not those characteristics may complement each other in certain situations. One interesting finding in [SKD20] is that the combination of Falcon-1024 and Dilithium-IV in one certificate chain resulted in a faster TLS handshake than using any of the algorithms alone. This is due to the difference in the sizes and the difference in the verify and sign speed of both schemes. In this section, the mixing of PQC schemes in certificate chains is explored.

3.5.1 Considerations for Mixed Post-Quantum X.509 Certificate Chains

This section explores mixed PQC algorithms in the X.509 PKI and focuses on the special role of the root certificate.

Large (Root) CA Certificates Root CA certificates play a special role since they are trust anchors that are already stored on the receiver's device. Thus, they are typically not part of a certificate chain when authenticating the sender. As an example, using large Rainbow keys in root certificates would not lead to increased bandwidth requirements since the root certificate is known to all participants. For constrained devices, it can add the burden to store the large keys on the flash memory and load them into RAM for validating an intermediate CA signature. However, in [Gon+21] it is already demonstrated that rainbowI-classic can stream in the public key and the signature on a device that utilizes only 8 kB of RAM and 8 kB of flash memory. If more than the root CA can be omitted from the certificate chain, the same reasoning applies to the intermediate CAs. For TLS, there is a draft[24] that proposes exactly this, in order to reduce the size of post-quantum certificate chains.

Stateful Signature Schemes in CA certificates Due to the concerns given in Section 3.4.7 regarding stateful signatures, we do not recommend them for general use. For XMSS and LMS, the state amounts to an integer, describing the currently active leaf node. For each made signature, it is increased, such that each leaf node is selected exactly once. CAs already employ strict guidelines and procedures and it is a very controlled environment, thus, additionally managing the state in this use case can be handled. Since public CAs already publish every issued certificate in the certificate transparency log, it would even be feasible to extract the state from the certificate transparency log by counting the number of issued certificates of a given CA certificate. When issuing certificates, care has to be taken to prevent race conditions, i.e., when two certificate requests are signed in a given time frame, the actual signing operation has to be serialized to prevent using the same state. When used correctly, XMSS and LMS have several advantages for CAs. Firstly, they are already standardized and their security assumptions are well understood. Secondly, Root CAs do not typically issue many certificates, thus a small parameter set with relatively small signatures can be

[24]`https://www.ietf.org/id/draft-kampanakis-tls-scas-latest-02.html`

chosen for Root CAs. CA certificates that issue a lot of end-entity certificates may have to choose larger parameter sets, though. Thirdly, XMSS and LMS are slow when signing, but fast when verifying. Slow signing speed is not an issue for the CAs during certificate issuance and is only done once, whereas the fast verification benefits the clients that verify the certificate chain. In conclusion, it might be a feasible strategy to deploy XMSS or LMS in Root CA and CA certificates, and other signature algorithms in the end-entity certificates, resulting in a mixed certificate chain which employs different algorithms in different certificates.

3.5.2 Mixing PQC Algorithms in TLS

In order to better understand the practical impact of mixing different PQC algorithms, an experiment on TLS 1.3 has been performed. The Open Quantum Safe project maintains an OpenSSL fork[25] and integrates many PQC algorithms into the OpenSSL TLS 1.3 implementation already. The integration into OpenSSL is described in detail in [CPS19b] and is also briefly described in this report in Section 3.4.3. With the help of the OpenSSL fork (at commit *ea1ab67*), a test suite has been created that measures TLS handshakes that utilize PQC algorithms, i.e., PQC certificate chains and PQC key exchanges.

Test Setup and Methodology

To systematically test the impact of PQC algorithms in TLS, first, a variety of certificate chains has been created. These certificate chains were then used in a TLS handshake and the impact on the handshake size and speed has been recorded. The setup is described in detail in the following.

Generating Certificate Chains Certificate chains of size three, containing a root-CA certificate, a sub-CA certificate, and an end-entity certificate are generated. For each of the three certificates, a different PQC signature algorithm can be chosen. However, in order to reduce the number of certificate chains, the root CA and the sub CA certificates are always generated with the same algorithm. To further reduce the number of certificates to a reasonable size, not all algorithms that the OpenSSL fork offers are considered. The list of considered signature algorithms is the following:

- dilithium2_aes, dilithium5_aes,
- falcon512, falcon1024,
- picnic3l1, picnic3l5,
- rainbowIIIclassic, rainbowVclassic, rainbowIIIcompressed, rainbowVcompressed,
- sphincssha256128fsimple, sphincssha256128ssimple, sphincssha256256srobust, sphincssha256256frobust.

[25] `https://github.com/open-quantum-safe/openssl`

This results in a total number of $14 \cdot 14 = 196$ different certificate chains. The selection criteria for the signature algorithms have been:

1. Different algorithm variants are only included if they have major differences, e.g., the SPHINCS$^+$ robust and simple variants have a large difference in runtime.
2. AES variants have been chosen when both AES and other variants are offered (e.g. dilithium2_aes instead of dilithium2).
3. To reduce the number of algorithms, only level 1 and level 5 variants have been chosen. This gives the minimum and the maximum impact of the respective algorithms. Dilithium is an exception, since only level 2 is offered instead of level 1. Further, since Rainbow's level 1 parameter set has been broken [Beu22] and subsequently removed from the OpenSSL fork, level 3 is considered instead.
4. The SPHINCS$^+$ Haraka and SHAKE256 variants have been omitted entirely, and only the SHA256 variant is included. This already amounts to four variants which already occupy more than a quarter of algorithms in the final list of considered signature algorithms.

Handshake Combinations A complete PQC handshake also utilizes PQC KEMs for the key exchange. The same criteria as for the signature algorithms are applied to the PQC key-exchange algorithms that the OpenSSL fork offers. The key exchange algorithms that are considered, are:

- bikel1, bikel3,
- kyber512, kyber1024,
- frodo640aes, frodo1344aes,
- hqc128, hqc256,
- ntru_hps2048509, ntru_hps40961229, ntru_hrss701, ntru_hrss1373, ntrulpr653, ntrulpr1277,
- sntrup653, sntrup1277,
- lightsaber, firesaber,
- sidhp434, sidhp751,
- sikep434, sikep751.

Note that SIKE/SIDH have been removed in more recent versions of the OpenSSL fork, due to a presented attack on the scheme.[26] This results in a total number of $14 \cdot 14 \cdot 22 = 4312$ different handshakes.

[26] `https://github.com/open-quantum-safe/openssl/pull/383`, also see `https://github.com/open-quantum-safe/liboqs/pull/1272`

Handshake Measurements For each of the 4312 handshakes, the handshake size, i.e., the data that is sent by the client and the data that is sent by the server, as well as the speed of completing the handshake is reported. The handshakes are left to the default behaviour of the respective OpenSSL tools. The server authenticates itself with one of the generated certificate chains and omits the root CA, i.e., only sends the sub-CA certificate and the end-entity certificate. The client does not authenticate itself. For the measurements that are described in the following paragraphs, the server will always be an instance of the OpenSSL *s_server* tool.

Determine Handshake Size To determine the size of the handshake, the OpenSSL *s_client* tool is used to perform a handshake. The output is parsed and the size of the handshake is recorded, divided by the data that is sent by the server and the data that is sent by the client.

Determine Handshake Speed To measure the handshake speed, the OpenSSL *s_time* tool is used to perform a handshake and measure its speed. Both, the server, and the client, are run locally on the same system. The system features an Intel i5-8400 CPU. Each of the handshake combinations is measured for at least 15 seconds. If there are less than 5 handshakes measured in total, the combination is tested again with an increased time, to ensure some minimum number of measured handshakes. Each combination is tested in two different network setups:

- Normal speed, i.e., no restrictions.
- The tool trickle[27] is used which is a simple traffic shaper with which the network speed is reduced to $256\,\mathrm{kB/s}$.

By comparing the unlimited with the limited results, some conclusions can be drawn with regards to the impact of slower connections and the sizes of the keys, ciphertexts, and signatures.

Results

This section describes the results of the previous measurements. For the rest of the section, the SPHINCS$^+$ variants will be abbreviated by omitting the *sha256* substring, since only SHA-256 variants are considered. Further, *sphincs* is abbreviated by *spx* to make it a bit shorter, E.g., *sphincssha256256frobust* will become *spx256frobust*.

Overview The total number of data sets is $3 \times (14 \cdot 14 \cdot 22) = 12936$, since we measure $14 \cdot 14 \cdot 22$ handshakes for speed, speed (with trickle), and size. Therefore, it is not feasible to display the data in full detail. First, an impression of the overall results is given by demonstrating each the three min, max, and median values. Table 3.14 depicts these values for the handshake size. For the handshake speeds, the same is depicted in Table 3.15 for the unlimited variant, and in Table 3.16 for the variant that limits speed with trickle.

[27]`https://github.com/mariusae/trickle`

Root CA	Sub CA	EE	KEX	HS Size	Client Sent	Server Sent
			Smallest			
falcon512	falcon512	falcon512	sidhp434	5596	911	4685
falcon512	falcon512	falcon512	sikep434	5614	911	4703
falcon512	falcon512	falcon512	sidhp751	6064	1145	4919
			⋮			
			Median			
spx256frobust	spx256frobust	spx256frobust	kyber512	152,631	1381	151,250
spx256frobust	spx256frobust	spx256srobust	hqc256	152,669	7826	144,843
spx256srobust	spx256srobust	picnic3l5	frodo1344aes	152,799	22,106	130,693
			⋮			
			Largest			
rainbowVclassic	rainbowVclassic	rainbowVclassic	frodo640aes	3,887,533	10,197	3,877,336
rainbowVclassic	rainbowVclassic	rainbowVclassic	hqc256	3,889,911	7826	3,882,085
rainbowVclassic	rainbowVclassic	rainbowVclassic	frodo1344aes	3,911,359	22,106	3,889,253

Table 3.14: The three smallest, the three median, and the three largest measured handshakes. Columns 1-4 depict the chosen signature and key-exchange algorithms. The *HS Size* (Handshake Size) column is the sum of the *Client Sent* and *Server Sent* columns which describe the amount of bytes that have been sent by each party.

Root CA	Sub CA	EE	KEX	Handshakes / s
		Slowest		
rainbowVclassic	rainbowVclassic	rainbowVclassic	sikep751	4.1
rainbowVcompressed	rainbowVcompressed	rainbowVclassic	sikep751	4.15
rainbowVclassic	rainbowVclassic	rainbowVcompressed	sikep751	4.23
		⋮		
		Median		
dilithium5_aes	dilithium5_aes	rainbowIIIclassic	sidhp434	29.62
spx256srobust	spx256srobust	rainbowIIIclassic	kyber1024	29.69
spx256srobust	spx256srobust	rainbowIIIclassic	ntru_hrss701	29.69
		⋮		
		Fastest		
dilithium2_aes	dilithium2_aes	dilithium2_aes	kyber1024	2115.05
dilithium2_aes	dilithium2_aes	dilithium2_aes	lightsaber	2141.47
dilithium2_aes	dilithium2_aes	dilithium2_aes	kyber512	2201.1

Table 3.15: The three slowest, the three median, and the three fastest measured handshakes. Columns 1-4 depict the chosen signature and key-exchange algorithms and Column 5 depicts the measured handshake speed.

Root CA	Sub CA	EE	KEX	Handshakes / s
		Slowest		
rainbowIIIclassic	rainbowIIIclassic	spx256srobust	frodo640aes	0.22
dilithium5_aes	dilithium5_aes	spx128ssimple	ntrulpr1277	1.68
rainbowVclassic	rainbowVclassic	rainbowVclassic	sikep751	3.9
		⋮		
		Median		
spx256srobust	spx256srobust	dilithium2_aes	sikep434	26.55
spx256frobust	spx256frobust	spx256frobust	ntru_hrss701	26.6
spx128fsimple	spx128fsimple	falcon1024	sikep434	26.65
		⋮		
		Fastest		
falcon512	falcon512	dilithium2_aes	ntru_hps2048509	1462.16
falcon512	falcon512	dilithium2_aes	hqc128	1488.0
falcon512	falcon512	dilithium2_aes	lightsaber	1537.84

Table 3.16: The three slowest, the three median, and the three fastest measured handshakes when measuring with *trickle* enabled (limited to 256 kB/s). Columns 1-4 depict the chosen signature and key-exchange algorithms and Column 5 depicts the measured handshake speed.

Fastest Handshake in the Unlimited Variant The fastest measurement for the unlimited variant is the combination of *dilithium2_aes* for all three certificates, and *kyber512* as key exchange algorithm and 2201.1 handshakes per second are measured. When enabling trickle, this combination achieves only the 65th fastest measured speed with 575.81 handshakes per second which is around 3 times slower than the fastest combination with trickle enabled. The handshake size of the *dilithium2_aes* and *kyber512* combination is 12,661 B and ranks at the 143th position.

Fastest Handshake with Trickle Enabled The fastest measurement for the trickle-enabled variant is the combination of *falcon512* for the root and sub CA certificates, *dilithium2_aes* for the EE certificate, and *lightsaber* as key exchange algorithm and 1537.84 handshakes per second are measured. In the unlimited variant, this combination achieves the 21th rank with 1888.73 handshakes per second The handshake size of the combination is 8532 B and ranks at the 34th position.

Smallest Handshake Size The smallest handshake is the combination of *falcon512* and *sidhp434* with a total handshake size of only 5596 B. Since the *sidhp434* computations are slow, the combination only ranks at position 1083 with 79.53 handshakes per second in the unlimited variant. With trickle enabled, it ranks at position 813 with 78.88 handshakes per second. It can be seen that the network speed does not make much of a difference in this case and the computation time dominates the handshake speed.

Falcon and Dilithium In [SKD20] it is shown that a combination of falcon and dilithium performs better than any of the schemes used alone. That is, combining falcon for root and intermediate CAs with dilithium for end-entity certificates, the measured TLS handshake speeds are faster than when the complete chain uses falcon or dilithium. In our experiments, we see the same effect with the trickle-enabled variant. That is, combining falcon with dilithium in the certificate chain yields faster handshakes than utilizing only one of the algorithms. It is interesting to see that while *falcon512* and *dilithium2_aes* are the fastest combination in the trickle-enabled variant, *dilithium2_aes*-only is the fastest combination in the unlimited variant. The tradeoff between computational speed and key/signature sizes seems to shift towards *dilithium2_aes* for faster networks.

4 D3.2 Hardening Measures

This chapter describes the hardening measures of our software implementations of WP3 work package. Section 4.1 gives an overview of current attacks on physical implementations. Lastly, in Section 4.2 we discuss the design and implementation of counter-measures in the software libraries developed in this work package. Several countermeasures, especially to defend against timing-attacks have also been described in Section 3.

4.1 Overview of Physical Attacks

The primary aspect regarding the security of cryptographic systems is their mathematical security. A mathematical model, however, does not encompass physical properties of a cryptographic implementation. In the real world, there are *side-channels* that are introduced due to the physical nature of concrete implementations. A side-channel is a physical information channel over which implementation-specific characteristics leak. That is, side-channels are observable physical side-effects that are caused by operations of implementations.

Such side-channels are for example timings, power consumption, electromagnetic fields, and other physical properties that can be observed and that are influenced by an implementation. The information leaks on side-channels can lead to *side-channel attacks*, which in turn may compromise the security of a cryptosystem. Physical side-channel security usually lies outside the consideration of mathematical security of cryptosystems, but can be prohibited with carefully implemented countermeasures. Countermeasures typically remove or reduce the correlation of secret values to data that is leaked through a side-channel.

Physical attacks can be broadly categorized by *active* and *passive* attacks. While passive attacks are performed by only monitoring side-channels, active attacks also include the physical manipulation of a device in order to trigger different behavior in the program execution. Many side-channels are only exploitable from a near physical distance. Timing attacks can, however, also be applied remotely.

In the following, some physical attacks and respective countermeasures are treated in more detail.

4.1.1 Timing Attacks

Timing side-channel attacks have first been described in 1996 by Kocher for RSA, DH, and DSS [Koc96]. It was shown that for RSA and Diffie-Hellman modular exponentiations using the private key leak secret key bits through a timing side-channel. That is, the execution length of the operation depends on bits of the private key, thus revealing a correlation to these bits. The attack can be carried out as a passive attacker (known-ciphertext) or as an active attacker (chosen-ciphertext). When enough data is collected, the private key bits can be derived from the correlation with the respectively measured timings.

In 2003 it was demonstrated by Boneh and Brumley that such attacks can be carried out from a remote attacker [BB03]. In one of the considered scenarios, the attacker resides in the same local area network as the victim. The victim in this case is an OpenSSL server whose private key was successfully extracted by the attacker. The attack utilizes timing leakages that depend on the private key in the former OpenSSL implementation.

Thus, it has become apparent that cryptographic implementations must include countermeasures to timing attacks, since they can be executed passively and even remotely. The two main methods to protect against timing attacks are *masking* and *constant-time code*.

Masking is a technique where secret values that may leak timing information are transformed before using them. That is, the algebraic value is randomly transformed in such a way that it can be undone after the computation is done. The attacker can then only correlate the leaked timings with the masked values, not revealing a correlation to the actual secret value. Masking can only be applied when the mathematical structure allows for it.

Constant-time code ensures that no timing leakages exist that reveal any secret data. In other words, the execution time does not depend on secret data. There may be variances in the execution time of the implementation but only if they depend on publicly known or non-sensitive data.

4.1.2 Simple Power Analysis (SPA) and Differential Power Analysis (DPA)

Kocher also first described *simple power analysis* (SPA) attacks and *differential power analysis* (DPA) attacks [KJJ99].

In general, power analysis attacks analyze *power traces* of a device. Such a trace is obtained by measuring the electrical current while a cryptographic operation is executed. Different instructions exhibit different power consumption patterns, thus correlate to instructions that are executed.

SPA typically directly interprets a trace. It can also be applied to the mean of multiple traces of the same operation with the same input data to reduce noise. As an example, conditional branches can be recognized in a trace, which may reveal secret key bits if the branching depends on the secret key. A concrete example for this are naively implemented square-and-multiply modular exponentiation methods. Squaring or multiplying can usually be distinguished in power traces and the execution pattern depends on single bits of the (secret) exponent and thus immediately reveals those bits.

Differential power analysis makes use of multiple traces of operations that may use differing input data. While the traces might contain too much noise to apply SPA, DPA applies statistical models to find the relation between power consumption and secret data. In comparison to SPA, DPA relies less on existing knowledge of the implementation.

We note that SPA and DPA can also be used to aid brute-force attacks. As an example, the knowledge of the Hamming weight of a secret key, or even of each of its single bytes, reduces the search space.

4.1.3 Fault Attacks

Fault attacks are active attacks that aim at disturbing instructions during the computation of cryptographic operations through one or more faults. Such an attack was fist proposed in the cryptographic context by Boneh, DeMillo, and Lipton in 1997 [BDL97].

Techniques used for inducing faults are, for instance, clock glitching, power glitching, or the physical manipulation of memory locations. With a carefully chosen point for inducing faults, the attacker may observe secret-dependent behavior in the following execution of the algorithm. For example, some cryptosystems use secret-dependent dummy operations in order to achieve constant-time behavior that thwarts timing attacks; Faulting such an operation still yields the correct output, while faulting a necessary operation results in a faulty output. Observing this output, e.g. via observing whether a key agreement succeeds, can therefore leak information on secret key bits.

Fault attacks can be divided into two categories: first-order fault attacks induce a single fault during a cryptographic computation, while higher order fault attacks allow multiple faults per run. Work on fault attacks usually sets up a detailed attacker model that is necessary for executing the respective attacks in practice.

A simple, but effective countermeasure against first-order fault attacks is given by redundant computations, such that the correctness of the execution can be verified. Furthermore, protected implementations refrain from using dummy operations. Instead, to protect against fault attacks, implementations should use outputs of all instructions at a later point in the computation, such that faulting any instruction leads to an invalid output. Similarly, memory access patterns should be designed in a way that does not keep any values in memory without further being used. Thus, this can prevent memory faults that try to detect whether variables are reused in later computations.

4.2 Implemented Counter-Measures

This section outlines implemented side-channel-attack counter-measures. In Section 4.2.1 we present the design and constant-time implementation of Classic McEliece. Section 4.2.2 shows the software implementation of XMSS on a Hardware Security Module, which effectively secures the private keys.

4.2.1 Classic McEliece Streaming in Constant-Time

The newly developed streaming approach for Classic McEliece that has been described in Section 3.3.1 is hardened against timing attacks. The code has been written to be executed in constant time to avoid timing issues. It therefore retains the constant-time security property from the original code. Since the code utilizes an LU decomposition, special care has been taken to implement the permutation in constant time. Branches are realized by masking the operations with all-zero or all-one masks, resulting in more (needless) computations but ensures taking the same execution path across different inputs.

4.2.2 XMSS Implementation on a Hardware Security Module

To mitigate attacks on the implementation, an XMSS implementation has been developed for a hardware security module (HSM). While many counter-measures or mitigations can be implemented in software, a tamper-resistant HSM can protect the software from many physical attacks.

Utimaco, a company located in Aachen (Germany), produces configurable HSMs and is an associated partner in the QuantumRISC project. It is possible to develop custom firmware modules that can then be loaded into the HSM. This makes it possible to extend the capabilities of the HSM, as in this case by providing a custom firmware module that implements the XMSS signature scheme. For writing such a module, Utimaco offers the CryptoServer SDK[1].

An XMSS implementation has been developed for the Utimaco SecurityServer Se Gen2 Se12[2] using the CryptoServer SDK. The implementation is based on the XMSS reference code which is available on Github[3]. Adjustments to the implementation were made to port the reference code implementation to the HSM.

In our implementation the keys are stored securely in the HSM and the public key digest serves as an index to access the keys. We provide management API functions to list, generate, delete, and view additional information about the keys. The additional information that can be obtained, contains, among other things, the number of signatures that can still be generated with the key. As XMSS is a stateful signature scheme with a maximum number of signatures, this is an important information.

Since for many applications, detached signatures are needed (e.g. for signing X.509 certificates), the API provides these instead of the "“signed-message-API”" that the reference code offers.

In the following, the performance of the XMSS operations on the HSM is illustrated. As expected, the key generation takes a long time for the parameter sets with a large tree depth. The parameter sets with tree depths greater than 16 are excluded and have not been evaluated. When a large number of signatures from a single key is needed, it can be worthwhile to consider the multi-tree variant of XMSS. In this variant the signature size can be increased in favor of more signatures, while keeping the key generation time low.

[1] `https://hsm.utimaco.com/products-hardware-security-modules/software-development-kit-sdk/cryptoserver-sdk/`

[2] The datasheet can be obtained at `https://hsm.utimaco.com/products-hardware-security-modules/general-purpose-hsm/securityserver-se-gen2/`

[3] `https://github.com/XMSS/xmss-reference`

Parameter Set			KeyGen	Sign	Signatures	Verify
Hash Function	n	Tree Height	in s	in s	Operations/s	in s
SHA2-256	256	10	9.19	0.0373	26.83	0.0068
SHA2-512	512	10	47.68	0.1748	5.72	0.0250
SHAKE-128	256	10	22.21	0.0823	12.14	0.0131
SHAKE-256	512	10	80.52	0.2899	3.45	0.0444
SHA2-256	256	16	588.27	0.0385	25.95	0.0066
SHA2-512	512	16	3049.81	0.1787	5.60	0.0256
SHAKE-128	256	16	1419.50	0.0919	10.88	0.0123
SHAKE-256	512	16	5099.83	0.2956	3.38	0.0425

Table 4.1: Performance of the XMSS Module on the Utimaco HSM

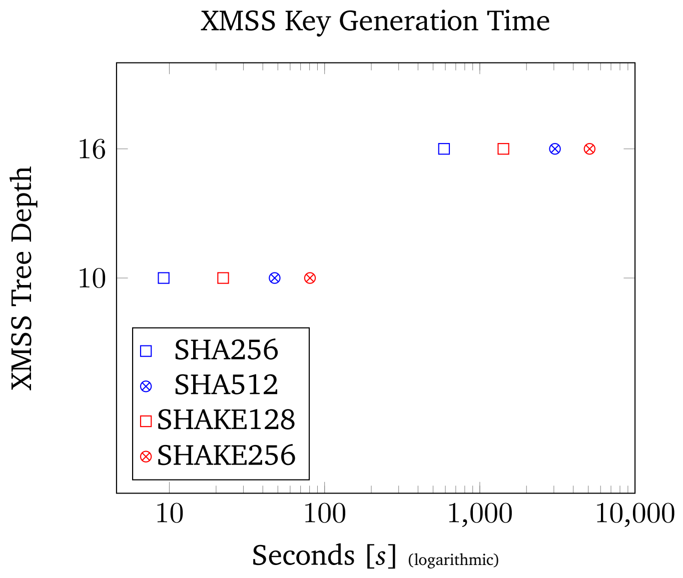

Figure 4.1: XMSS Key Generation Timings on the Utimaco HSM

4.2.3 Timing Attack and Countermeasure on the Rejection Sampling of HQC and BIKE

Although round 3 candidates of the NIST PQC competition have already been intensively vetted with regard to side-channel attacks, one important attack vector has hitherto been missed: PQ schemes often rely on *rejection sampling* techniques to obtain pseudorandomness from a specific distribution. In [Guo+22], we reveal that rejection sampling routines that are seeded with secret-dependent information and leak timing information result in practical key recovery attacks in the code-based key encapsulation mechanisms HQC and BIKE.

Both HQC and BIKE have been selected as alternate candidates in the third round and continued in the fourth round of the NIST competition, which puts them on track for getting standardized separately to the finalists. They have already been specifically hardened with constant-time decoders to avoid side-channel attacks. However, in our work, we show novel timing vulnerabilities in both schemes: (1) Our secret key recovery attack on HQC requires only approx. $866{,}000$ idealized decapsulation timing oracle queries in the 128-bit security setting. It is structurally different from previously identified attacks on the scheme: Previously, exploitable side-channel leakages have been identified in the BCH decoder of a previously submitted HQC version, in the ciphertext check as well as in the pseudorandom function of the Fujisaki-Okamoto transformation. In contrast, our attack uses the fact that the rejection sampling routine invoked during the deterministic re-encryption of the decapsulation leaks secret-dependent timing information, which can be efficiently exploited to recover the secret key when HQC is instantiated with the (now constant-time) BCH decoder, as well as with the RMRS decoder of the current submission. (2) From the timing information of the constant weight word sampler in the BIKE decapsulation, we demonstrate how to distinguish whether the decoding step is successful or not, and how this distinguisher is then used in the framework of the GJS attack to derive the distance spectrum of the secret key, using 5.8×10^7 idealized timing oracle queries. We further discusses possible countermeasures and their limits. Finally, Nicolas Sendrier proposes a more efficient countermeasure to our attack in [Sen21] which is now integrated in the BIKE and HQC reference implementations.

Bibliography

[Alk+20] Erdem Alkim et al. “ISA Extensions for Finite Field Arithmetic”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (2020). `https://tches.iacr.org/index.php/TCHES/article/view/8589`, pp. 219–242. ISSN: 2569-2925. DOI: `10.13154/tches.v2020.i3.219-242`.

[Ara+17] Nicolas Aragon et al. *BIKE – Bit Flipping Key Encapsulation*. `https://bikesuite.org/`. 2017.

[BB03] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *USENIX Security 2003: 12th USENIX Security Symposium*. USENIX Association, Aug. 2003.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. “On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)”. In: *Advances in Cryptology – EUROCRYPT’97*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, Heidelberg, May 1997, pp. 37–51. DOI: `10.1007/3-540-69053-0_4`.

[Beu22] Ward Beullens. *Breaking Rainbow Takes a Weekend on a Laptop*. Cryptology ePrint Archive, Report 2022/214. `https://eprint.iacr.org/2022/214`. 2022.

[BGJ22a] Alison Becker, Rebecca Guthrie, and Michael J. Jenkins. *Non-Composite Hybrid Authentication in PKIX and Applications to Internet Protocols*. Internet-Draft draft-becker-guthrie-noncomposite-hybrid-auth-00. Work in Progress. Internet Engineering Task Force, Mar. 2022. 10 pp. URL: `https://datatracker.ietf.org/doc/draft-becker-guthrie-noncomposite-hybrid-auth/00/`.

[BGJ22b] Alison Becker, Rebecca Guthrie, and Michael J. Jenkins. *Related Certificates for Use in Multiple Authentications within a Protocol*. Internet-Draft draft-becker-guthrie-cert-binding-for-multi-auth-01. Work in Progress. Internet Engineering Task Force, June 2022. 11 pp. URL: `https://datatracker.ietf.org/doc/draft-becker-guthrie-cert-binding-for-multi-auth/01/`.

[CC21a] Matt Campagna and Eric Crockett. *Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS)*. Internet-Draft draft-campagna-tls-bike-sike-hybrid-07. Work in Progress. Internet Engineering Task Force, Sept. 2021. 17 pp. URL: `https://datatracker.ietf.org/doc/draft-campagna-tls-bike-sike-hybrid/07/`.

[CC21b] Ming-Shing Chen and Tung Chou. “Classic McEliece on the ARM Cortex-M4”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.3 (2021), pp. 125–148. DOI: 10.46586/tches.v2021.i3.125-148. URL: https://tches.iacr.org/index.php/TCHES/article/view/8970.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. “Optimizing BIKE for the Intel Haswell and ARM Cortex-M4”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.3 (2021), pp. 97–124. DOI: 10.46586/tches.v2021.i3.97-124. URL: https://tches.iacr.org/index.php/TCHES/article/view/8969.

[Cel+22] Sofia Celi et al. *KEM-based Authentication for TLS 1.3*. Internet-Draft draft-celi-wiggers-tls-authkem-01. Work in Progress. Internet Engineering Task Force, Mar. 2022. 25 pp. URL: https://datatracker.ietf.org/doc/draft-celi-wiggers-tls-authkem/01/.

[Che+22] Ming-Shing Chen et al. “Carry-Less to BIKE Faster”. In: *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Springer, 2022, pp. 833–852. DOI: 10.1007/978-3-031-09234-3_41. URL: https://doi.org/10.1007/978-3-031-09234-3_41.

[CPS19a] Eric Crockett, Christian Paquin, and Douglas Stebila. *Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH*. Cryptology ePrint Archive, Paper 2019/858. https://eprint.iacr.org/2019/858. 2019. URL: https://eprint.iacr.org/2019/858.

[CPS19b] Eric Crockett, Christian Paquin, and Douglas Stebila. *Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH*. Cryptology ePrint Archive, Report 2019/858. https://eprint.iacr.org/2019/858. 2019.

[Das+20] Bhargav Das et al. “PQFabric: A Permissioned Blockchain Secure from Both Classical and Quantum Attacks”. In: *CoRR* abs/2010.06571 (2020). arXiv: 2010.06571. URL: https://arxiv.org/abs/2010.06571.

[Gon+21] Ruben Gonzalez et al. “Verifying Post-Quantum Signatures in 8 kB of RAM”. In: *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*. Ed. by Jung Hee Cheon and Jean-Pierre Tillich. Springer, Heidelberg, 2021, pp. 215–233. DOI: 10.1007/978-3-030-81293-5_12.

[Guo+22] Qian Guo et al. “Don’t Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* Vol. 2022. 3. 2022, pp. 223–263. DOI: 10.46586/tches.v2022.i3.223-263. URL: https://doi.org/10.46586/tches.v2022.i3.223-263.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology – CRYPTO’99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[KK18] Franziskus Kiefer and Kris Kwiatkowski. *Hybrid ECDHE-SIDH Key Exchange for TLS*. Internet-Draft draft-kiefer-tls-ecdhe-sidh-00. Work in Progress. Internet Engineering Task Force, Nov. 2018. 13 pp. URL: https://datatracker.ietf.org/doc/draft-kiefer-tls-ecdhe-sidh/00/.

[Koc96] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Advances in Cryptology – CRYPTO’96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[Mas+22] Jake Massimo et al. *Algorithms and Identifiers for Post-Quantum Algorithms*. Internet-Draft draft-massimo-lamps-pq-sig-certificates-00. Work in Progress. Internet Engineering Task Force, July 2022. 12 pp. URL: https://datatracker.ietf.org/doc/draft-massimo-lamps-pq-sig-certificates/00/.

[MM22] Marcel Müller and Michael Meyer. “QuantumRISC WP2 Report: Analysis and Optimization of PQC schemes”. 2022. URL: https://quantumrisc.org/results/quantumrisc-wp2-report.pdf.

[Noa+20] David Noack et al. *QuantumRISC WP1 Report: Use Cases and Requirements*. 2020. URL: https://quantumrisc.org/results/quantumrisc-wp1-report.pdf.

[NRW21] Ruben Niederhagen, Johannes Roth, and Julian Wälde. *Streaming SPHINCS+ for Embedded Devices using the Example of TPMs*. Cryptology ePrint Archive, Report 2021/1072. https://ia.cr/2021/1072. 2021.

[OG22] Mike Ounsworth and John Gray. *Composite KEM For Use In Internet PKI*. Internet-Draft draft-ounsworth-pq-composite-kem-00. Work in Progress. Internet Engineering Task Force, July 2022. 24 pp. URL: https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-kem/00/.

[OGM22] Mike Ounsworth, John Gray, and Serge Mister. *Composite Encryption For Use In Internet PKI*. Internet-Draft draft-ounsworth-pq-composite-encryption-01. Work in Progress. Internet Engineering Task Force, Feb. 2022. 22 pp. URL: https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-encryption/01/.

[OMG22] Mike Ounsworth, Serge Mister, and John Gray. *Explicit Pairwise Composite Keys For Use In Internet PKI*. Internet-Draft draft-ounsworth-pq-explicit-composite-keys-01. Work in Progress. Internet Engineering Task Force, Feb. 2022. 15 pp. URL: https://datatracker.ietf.org/doc/draft-ounsworth-pq-explicit-composite-keys/01/.

[OP22] Mike Ounsworth and Massimiliano Pala. *Composite Signatures For Use In Internet PKI*. Internet-Draft draft-ounsworth-pq-composite-sigs-07. Work in Progress. Internet Engineering Task Force, June 2022. 23 pp. URL: `https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-sigs/07/`.

[OPK22] Mike Ounsworth, Massimiliano Pala, and Jan Klaußner. *Composite Public and Private Keys For Use In Internet PKI*. Internet-Draft draft-ounsworth-pq-composite-keys-02. Work in Progress. Internet Engineering Task Force, June 2022. 30 pp. URL: `https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-keys/02/`.

[Qrw] “QuantumRISC WP4 Report”. 2022. URL: `https://quantumrisc.org/results/quantumrisc-wp4-report.pdf`.

[RKK21] Johannes Roth, Evangelos Karatsiolis, and Juliane Krämer. *Classic McEliece Implementation with Low Memory Footprint*. Cryptology ePrint Archive, Report 2021/138. `https://ia.cr/2021/138`. 2021.

[Sen21] Nicolas Sendrier. *Secure Sampling of Constant-Weight Words – Application to BIKE*. Cryptology ePrint Archive, Report 2021/1631, 20211217:142141 (posted 1639750901 17-Dec-2021 14:21:41 UTC). `https://eprint.iacr.org/2021/1631/20211217:142141`. 2021.

[SFG22] Douglas Stebila, Scott Fluhrer, and Shay Gueron. *Hybrid key exchange in TLS 1.3*. Internet-Draft draft-ietf-tls-hybrid-design-04. Work in Progress. Internet Engineering Task Force, Jan. 2022. 20 pp. URL: `https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/04/`.

[SKD20] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. *Post-Quantum Authentication in TLS 1.3: A Performance Study*. Cryptology ePrint Archive, Report 2020/071. `https://eprint.iacr.org/2020/071`. 2020.

[SS17] John M. Schanck and Douglas Stebila. *A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret*. Internet-Draft draft-schanck-tls-additional-keyshare-00. Work in Progress. Internet Engineering Task Force, Apr. 2017. 10 pp. URL: `https://datatracker.ietf.org/doc/draft-schanck-tls-additional-keyshare/00/`.

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. “Post-quantum TLS without handshake signatures”. In: *Proc. 27th ACM Conference on Computer and Communications Security (CCS) 2020*. ACM, 2020. DOI: `10.1145/3372297.3423350`.

[SSW21] Peter Schwabe, Douglas Stebila, and Thom Wiggers. *More efficient post-quantum KEMTLS with pre-distributed public keys*. Cryptology ePrint Archive, Paper 2021/779. `https://eprint.iacr.org/2021/779`. 2021. URL: `https://eprint.iacr.org/2021/779`.

[ST20] National Institute of Standards and Technology. *Recommendation for Stateful Hash-Based Signature Schemes*. Tech. rep. NIST Special Publication 800-208. Washington, D.C.: U.S. Department of Commerce, 2020. DOI: `10.6028/NIST.SP.800-208`.

[Ste+] Douglas Stebila et al. *Open Quantum Safe Project*. URL: `https://openquantumsafe.org`.

[Tur+22] Sean Turner et al. *Algorithm Identifiers for NIST's PQC Algorithms for Use in the Internet X.509 Public Key Infrastructure*. Internet-Draft draft-turner-lamps-nist-pqc-kem-certificates-01. Work in Progress. Internet Engineering Task Force, Mar. 2022. 8 pp. URL: `https://datatracker.ietf.org/doc/draft-turner-lamps-nist-pqc-kem-certificates/01/`.

[Vre+22] Christine van Vredendaal et al. *Quantum Safe Cryptography Key Information*. Internet-Draft draft-uni-qsckeys-01. Work in Progress. Internet Engineering Task Force, May 2022. 41 pp. URL: `https://datatracker.ietf.org/doc/draft-uni-qsckeys/01/`.

[Why+17] William Whyte et al. *Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3*. Internet-Draft draft-whyte-qsh-tls13-06. Work in Progress. Internet Engineering Task Force, Oct. 2017. 19 pp. URL: `https://datatracker.ietf.org/doc/draft-whyte-qsh-tls13/06/`.