INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

Practicability of XMSS on Javacards

Thomas Kagermeier

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

Practicability of XMSS on JavaCards

Thomas Kagermeier

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller

Betreuer: Sophia Grundner-Culemann
Dr. Tobias Guggemos
Stefan-Lukas Gazdag (genua GmbH)

Abgabetermin: 28. Juli 2020

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 28. Juli 2020

. . . .
(Unterschrift des Kandidaten)

Abstract

Advancements in the field of quantum computing pose a threat to cryptographic systems used today. As a result, alternatives for all kinds of cryptographic algorithms need to be found, with digital signatures being one of them. However, with the transition to quantum-secure signature schemes a number of challenges arise. The increased computational effort necessary for signature generation and the associated key sizes are especially problematic when executed on devices with limited resources like SmartCards. Since they are a crucial building block in modern security systems, it is important to bring quantum-secure schemes onto them. A special kind of SmartCard are JavaCards. While not being able to provide the same level of performance as native cards programmed in C or Assembler, the usage of the Java Virtual Machine allows applications to run manufacturer and hardware independent. This can be an important factor when it comes to widespread distribution of a new quantum-secure signature scheme.

A promising candidate regarding quantum-secure signature algorithms is XMSS. This hash based scheme solely relies on the security of the underlying hash function. However it comes with a high computational cost of repeated hash function evaluations. The goal of this thesis is to implement this scheme on a JavaCard and analyze its real-world practicability. The implementation uses the BDS tree traversal algorithm, providing a trade-off between memory usage and runtimes. By decoupling the computationally most expensive part of XMSS into an independent subroutine, operations necessary for the creation of signatures are minimized. The provided API allows signature generation with minimal communication between SmartCard and host. A parameter set optimized for fast runtimes is found and the resulting measurements are compared to requirements derived from possible use-cases.

The results show that real-world applicability can only be achieved to a limited degree with runtimes significantly longer than the ones of conventional signature algorithms.

Contents

1 Introduction 1

2 Background & Related Work 3
2.1 Digital Signatures . . . 3
2.2 Public-Key Cryptosystems . . . 3
2.2.1 Classical Digital Signatures . . . 4
2.2.2 RSA . . . 4
2.2.3 DSA and ECDSA . . . 5
2.2.4 ECDSA . . . 6
2.3 Quantum Computers . . . 7
2.3.1 General Concept . . . 7
2.4 Post-Quantum security . . . 8
2.4.1 Shor's Algorithm . . . 8
2.4.2 Grover . . . 9
2.5 Hash functions . . . 9
2.6 Hash-based signatures . . . 10
2.6.1 Winternitz One-Time Signature Scheme . . . 10
2.6.2 Forward secrecy . . . 12
2.6.3 XMSS - Extended Merkle Signature Scheme . . . 13
2.6.4 L-tree construction . . . 13
2.6.5 XMSS Hash Tree construction . . . 13
2.6.6 XMSS signature generation . . . 14
2.6.7 XMSS signature verification . . . 14
2.6.8 XMSS^{MT} - Multi-tree XMSS . . . 15
2.7 Tree traversal techniques . . . 16
2.7.1 Notation and general workflow . . . 17
2.7.2 Basic optimization strategies . . . 17
2.8 BDS Tree traversal . . . 18
2.8.1 BDS Scheduler . . . 19
2.8.2 Cost of subtree switches . . . 19
2.9 SmartCards . . . 20
2.9.1 Internal Components . . . 20
2.9.2 SmartCard Interface . . . 21
2.9.3 APDU - Application Protocol Data Unit . . . 22
2.9.4 Javacard Operating System . . . 24
2.9.5 JavaCard limitations . . . 25
2.9.6 Benefits of SmartCard and JavaCard based XMSS . . . 25
2.10 Related Work . . . 26
2.10.1 XMSS with off-card key generation . . . 26
2.10.2 Security risks of AES-based hash functions . . . 26

2.10.3 Forward Secure Signatures on Smart Cards 27
2.10.4 XMSS on JavaCards 27
2.10.5 Post-quantum security on other constrained devices 28

3 Methodology and Concept 31
3.1 Requirement Analysis 31
3.2 Requirements for the implementation 32
3.3 Advantages of JavaCard 32
3.4 The Hardware choice 32

4 Implementation 33
4.1 Development steps 33
4.2 JavaCard programming guidelines and best practises 34
4.3 Description of the final implementation 35
4.3.1 Provided API 37
4.4 Limitations 37
4.5 Test and Verification 38
4.5.1 Variations in WOTS+ signing times 38
4.5.2 Verification 39

5 Evaluation 41
5.1 General JavaCard performance 41
5.2 Splitting the signing and preparation steps 42
5.3 Finding appropriate parameter sets 42
5.3.1 Winternitz parameter w 42
5.3.2 Impact of parameter k 43
5.3.3 Subtree height and number 44
5.4 Test of larger trees 47
5.5 Conclusion of the Evaluation 48

6 Analysis of possible use cases 51
6.1 Criteria for use-case analysis 51
6.1.1 Number of possible signatures 51
6.1.2 Signature Size 51
6.1.3 Runtime 51
6.1.4 Security 52
6.1.5 Usability 52
6.2 E-mail signing 52
6.2.1 Current state 53
6.2.2 Requirement Definition 53
6.2.3 Client vs. Server based signing 54
6.2.4 Client based signing 54
6.2.5 Server based signing 56
6.2.6 Key-server based signing 59
6.2.7 Conclusion for the e-mail use-case 59
6.3 Authentication during VPN key negotiation 59
6.3.1 Digital signatures in IPsec authentication 60

6.3.2 IKEv2 handshake 60
6.3.3 Requirements for a signature scheme in IKEv2 61
6.3.4 Description and analysis of the use-case 62
6.3.5 Conclusion for the VPN use-case 63
6.4 Summary 63

7 Conclusion and Future Work **65**
7.1 Conclusion 65
7.2 Future Work 65
7.2.1 Distributed XMSS leaf computation 66

List of Figures **71**

Bibliography **73**

1 Introduction

In October 2019 a group of researchers working at Google published a paper with the title *Quantum supremacy using a programmable superconducting processor* [AAB+19]. Whether the proclaimed quantum supremacy was really achieved in this case is up to debate, as only a couple of days later IBM responded with a statement, remarking that the proclaimed runtime of the algorithm on classical computers was overestimated significantly[1]. Regardless if quantum supremacy was achieved in this case, this paper shows that quantum computers are not only an active research field but also that advancements are being made. While a large number of areas in the economic and scientific sector could benefit from widely available large scale quantum computers, they also pose a threat to current IT security infrastructures. At least since the publication of Shor's algorithm [Sho94] the advancements in the creation of large-scale quantum computers are also seen as problematic. Widespread cryptographic schemes like RSA and DSA rely on the hardness of mathematical problems. Shor showed that if a sufficiently powerful quantum computer existed, these problems could be solved efficiently, thus rendering the schemes insecure. As a result, new types of cryptographic schemes must be introduced in order to enable secure communication for the future, like the world is used to today. The field dedicated to researching cryptographic schemes that remain secure, even if large scale quantum computers existed, is called post-quantum cryptography.

Two factors important for secure communication are authenticity and integrity, or in other words, knowing one is talking to the right communication partner and that the messages where transmitted unaltered. These properties can be ensured by the usage of digital signatures. One type of digital signatures that are considered secure from quantum computer attacks are hash-based signatures, whose security is relying on the hardness of breaking cryptographic hash functions. The research project *Quantencomputer-resistente Signaturverfahren für die Praxis* (squareUP) [2] was a three year research project dedicated to hash-based signatures. In May 2018 a Request for Comments (RFC) was published as a result of this project, describing the hash-based signature scheme XMSS [HBG+18]. The introduction of this scheme also introduces new challenges. Compared to conventional schemes like RSA and DSA, creating XMSS signatures requires significantly more computational steps. This can be problematic when it is required to be executed on devices with limited resources. However SmartCards, an example for such a device, are in wide use today when a high level of security is required. They can act as a possession factor in multi-factor authentication, are used as physical keys to unlock doors and are able to securely store digital keys. However, their hardware is not designed to perform demanding tasks but rather to execute small applications. Nonetheless, implementing such a scheme on a SmartCard is an important step towards making post-quantum signatures widely available and applicable in the real world. The usage of a secure hardware token like a SmartCard is required in a number of cases where highly confidential data is handled and the security of a cryptographic scheme or key storage inside software is not considered secure enough.

[1] https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy, visited 24th of July
[2] https://www.square-up.org, visited 24th of July

The focus of this thesis is to analyze whether the computational demands of XMSS and the limited resources provided by SmartCards, or more specifically, JavaCards can be combined. This type of card offers relevant advantages when it comes to widespread distribution of the scheme. They allow manufacturer independent execution of applets and are widely available with different hardware configurations, thus preventing special requirements depending on the card model.

Outline

This theses is structured as follows: In Chapter 2 the necessary background is discussed, including conventional signature schemes and the attacks quantum computers could perform on them. Afterwards the XMSS signature scheme is described before the properties of SmartCards are introduced. Section 2.10 presents other approaches of implementing XMSS on SmartCards. Chapter 3 describes the methodology of the approach and presents a reasoning behind software design and hardware choices before the implementation is described in Chapter 4. In Chapter 5 the evaluation results are presented. In Chapter 6 an analysis of a number of use cases and the practicability of the scheme is discussed. Chapter 7 finally shows possible future work and gives a conclusion.

2 Background & Related Work

The following chapter first gives an introduction to digital signatures in general and then describes attacks on conventional cryptographic schemes through quantum computers. Afterwards the necessary background for this work is introduced, including the implemented signature scheme as well as the hardware.

2.1 Digital Signatures

In general, digital signatures are used to verify the authenticity and integrity of a message and ensure the signer can not repudiate it. Similar to written signatures on a piece of paper a digital signature can be used to prove a message or other piece of data originated from the signing author. In addition, a digital signature also ensures the integrity of a message. In a simple example Alice composes a message and sends it to Bob. Now how can Bob verify the Message actually came from Alice and not a malicious party in between? Using a digital signature scheme Alice can sign the message with a signature only she is able to generate. Any receiver, including Bob, can now verify that the message could solely be created by Alice. If an attacker had changed the contents of the message, he would not have been able to generate a valid signature, as only Alice knows the required key to do so. This way, the integrity of the message is ensured.

2.2 Public-Key Cryptosystems

The following section is intended to provide a general introduction and overview over public key cryptosystems. Unlike symmetrical encryption systems, public-key systems do not rely on one common shared secret to communicate privately. Instead a key pair, consisting of the secret key sk and the public key pk is used. While pk can be obtained from anyone, sk needs to be only known by the owner of the key pair. Additionally it must not be possible to guess one key from knowing the other.

If Alice wants to send Bob a private message, Alice can use Bob's pk to encrypt the data. This cypher can now only be read by Bob, as only he knows sk, the key needed for decryption. In the context of digital signatures, a similar system can be used. This time, consider Bob wants to sign a file published publicly on his website. In most cases, he calculates the hash value of the file in order to reduce its size from an arbitrary length to a fixed size. He then uses his sk to sign this value and publishes this signature alongside the file. Everyone who wants to check the signature can then use Bob's pk to decrypt the cypher, thus generating the hash value of the published file, which can in turn be compared to an independently generated hash value, that was calculated from the downloaded file. As only Bob knows his sk, only he was able to generate the cypher that is decryptable using pk, thereby proving the authenticity and integrity of the data. [KL07]

2.2.1 Classical Digital Signatures

Classical digital signature schemes as they are used today in many applications rely on the hardness of mathematical problems. An example for such a problem is integer factorization. While it is a simple operation to multiply two prime numbers, there is no algorithm that is able to efficiently calculate the prime factor of a sufficiently large natural number. The actual threshold of what can be considered a sufficiently large number has changed since the introduction of RSA in 1977. In 1993 it was estimated that the factorization of a 129-digit number would at least take 5000 MIPS years [DDLM94]. On February 2, 1999 RSA-140 was successfully factorized in about 2100 MIPS years. Later the same year RSA-155, a 512bit number was factorized[CDL+99][CDL+00]. These advancements can be attributed to the discovery of new algorithms and more powerful computers. As 512-bit numbers where widely used at the time, this breakthrough shows a problem of many cryptographic functions. Given enough time or resources, what is considered state of the art today, can be vulnerable in the near future. To illustrate this further, today a 512-bit prime number can be factorized in less than four hours [VCL+15]. Today, the BSI recommends RSA key lengths of at least 2000bit for a sufficient security level[BSI20].

2.2.2 RSA

In the following section the functionality of RSA [RSA78] will be described. Its security relies on the difficulty of finding the prime factors of large integers.

RSA Key generation

Two large prime numbers, denoted p and q are chosen and their product $n = pq$ is calculated. Next, an integer e is chosen where:

$$1 < e < \phi \text{ with } \phi = (p-1)\,(q-1) \text{ and } e \text{ being a coprime to } \phi$$

Finally a value d needs to be calculated, which satisfies the following:

$$d = \frac{1}{e} mod_{\phi}\,(n)$$

The private exponent d together with the prime number n is used to encrypt messages. Decryption is possible using the public exponent e.

RSA signing and verification

In order to create a signature, the signer calculate the hash value of his or her message m and raises it to the power of of the private exponent d:

$$s \leftarrow m^d (mod\, N)$$

The receiver is then able to verify the signature by transforming it back to the original hash value by raising it to the power of the public exponent e:

$$h(m) \leftarrow s^e (mod\, N)$$

RSA Security

RSA implemented exactly this way does not provide sufficient security for actual usage, as other attacks than factoring n are possible in this case[KL07]. It is common to choose 3, 17, or 65537 as the value e [Sma15]. Smaller values are often used on constrained devices like SmartCards, although this has a negative impact on security. In this case the Chinese Remainder Theorem can be used to decrypt the message directly without needing to factorize n [Reg04]. A similar problem exists when the public exponent is too small[Cop97]. While many other optimizations to the implementation as well as possible attacks exist, RSA in general is considered to be secure as long as prime number factorization remains a hard problem.

2.2.3 DSA and ECDSA

While RSA can be used for signatures and encryption, Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) are specifically used for digital signatures. DSA is based on the discrete logarithm problem (DLP) and is standardized by the National Institute of Standards and Technology (NIST) since 1991. The draft for the next Federal Information Processing Standard Publication (FIPS), FIPS 186-5 [1], no longer lists standard DSA as an approved algorithm for digital signatures. For illustrating the use of the DLP in digital signatures, DSA is described in the following section nonetheless.

Discrete Algorithm Problem

The following definition of the DLP is taken from [Sma15]:
Let $(G, \cdot)$ be a finite abelian group of prime order q, such as a subgroup of the multiplicative group of a finite field or the set of points on an elliptic curve over a finite field. The discrete logarithm problem, or DLP, in G is: given $g, h \in G$, find an integer $x \in [0, ..., q)$ (if it exists) such that $g^x = h$. The mathematical problem is therefore to find x so that $x = dlog_g(h)$.

DSA Key generation

The following steps are performed to generate pk and sk [DK15]:

- Choose two prime numbers p and q, so that q is a divisor of $p - 1$. The last FIPS that specified DSA [2] allows the following bit lengths for p and q :
  (1024,160), (2048,224), (2048,256), (3072,256)

- To get an element g of order q from the group G. This is done by selecting random elements $h \in \mathbb{Z}_p^*$ until $g := h^{(p-1)/q} \neq [1]$.

- Choose a random integer x with $1 \leq x \leq q - 1$

- sk: (p,q,g,x)
  pk: (p,q,g,y) with $y := g^x$

[1] https://csrc.nist.gov/publications/detail/fips/186/5/draft
[2] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

DSA signing

The signer calculates the hash of a message m using a cryptographic hash function. Its output length affects the security of DSA as well as the length of p and q and should therefore be chosen to produces results of at least q bit. The output is then signed the following way:

- select random integer k with $1 \leq k \leq q-1$
- calcualte r and s:
  $r := (g^k \, mod \, p) \, mod \, q$
  $s := k^{-1}(m + rx) \, mod \, q$, if $s = 0$, choose new k.
- m, r, s is the signed message

DSA verification

The verifier who has the pk (p, q, g, y) and received the signature m, r, s can check its validity though the following steps:

- check the following conditions:
  $1 \leq r \leq q-1$
  $1 \leq s \leq q-1$
- compute $t = s^{-1} \, mod \, q$
- compute $v = ((g^m y^r)^t mod \, p) \, mod \, q$
- The signature is valid if $v = r$

2.2.4 ECDSA

ECDSA is a variant of DSA that works on elliptic curves instead of finite fields. Using classical computers, finding discrete logarithms on elliptic curves is harder than on finite fields as well as solving the integer factorization problem. It therefore offers the advantage of shorter signatures compared to DSA while maintaining the same security level. Every problem relying on the DLP can easily be modified to work on elliptic curves. Instead defining p and q as elements in a finite field.

ECDSA and DSA security

While finding x is easy for some G, in other groups the problem is infeasible to solve computationally. For hard cases a number of algorithms exist to solve the DLP. These include various versions of the baby step / giant step algorithm with a complexity of $\mathcal{O}(\sqrt{n}) log \, n$ [CLP05, MD18] and the Silver–Pohlig–Hellman Algorithm with a time complexity of $\mathcal{O}(\sqrt{n})$. Similar to the RSA algorithm however, there is no known algorithm that efficiently solves the DLP on a classical computer.

2.3 Quantum Computers

In order to provide a better insight into the threads that Quantum Computers (QC) pose to conventional signature algorithm, the following chapter describes their basic functionalities and their possible applications. As their name implies, quantum computers are based on quantum mechanical principles. While quantum effects are problematic for the advancement of classical computers with shrinking transistor sizes being susceptible to quantum tunneling [MG15], quantum mechanics can also be used to solve previously non-computable problems.

2.3.1 General Concept

Conventional computers work with bits. A bit can be in one of two possible states, either 0 or 1. A quantum computer works with so called qubits. Those are not limited to being in one state at a time but can be in both states at once by an arbitrary proportion. This state is called superposition.

Superposition

Superposition is only possible as long as the state is not measured. As soon as the value of a qubit is sampled, the qubit is collapsed to either state 0 or 1. The actual state of a qubit at the point in time it is measured depends on a probability. This probability is influenced by multiple factors. A qubit can be represented the following way [KGHGC19] :

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

where $\alpha, \beta \in \mathbb{C}$ and $\langle\psi|\psi\rangle$, satisfying

$$|\alpha|^2 + |\beta|^2 = 1$$

Measuring the state results in $|0\rangle = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ with a probability of $|\alpha|^2$ and with a probability of $|\beta|^2$ in $|1\rangle = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$.

Quantum circuits

The probability can be influenced by quantum logic gates which are the equivalent to logic gates in conventional digital circuits. Quantum gates are the building blocks of quantum circuits. There are different kinds of gates. The Hadamard gate for example is used to transform a qubit in a collapsed state back to superposition with equally proportioned probabilities for α and β. Other types of gates are Pauli, not and swap gates. By connecting gates more complex operations are possible.

Entanglement

A series of n classical bits can be in one of 2^n different states while the same number of qubits can be in all of those 2^n states at once. Unlike in regular physics, where individual entities in a systems have a separate state, in quantum mechanics one qubit can be influenced by others. This phenomenon is called entanglement and is useful in quantum computers. For example, from measuring on qubit and collapsing its superposition in the process, the state of an entangled qubit can be determined without ending its superposition.

2.4 Post-Quantum security

2.4.1 Shor's Algorithm

In 1997 Peter Short introduced an algorithm that, given a powerful enough QC, could find factors of large prime numbers and compute discrete logarithms [Sho97]. As described in the Subsection 2.2.2 and Subsection 2.2.3, RSA and DSA rely exactly on the hardness of these problems.

General functionality and Integer Factorization

There is a number of algorithms for factorizing integers on conventional computers [Pol75, Hia14]. Although these algorithms are able to deterministically perform factorization, in practice this is not computationally feasible. Other methods like the Fermat Factorization Method relies on p and q to be close to one another [Kob94]. In order to find the factors p and q of a large prime number N, shorts algorithm starts with a random number $g, 1 < g < N$. The algorithm does not necessarily need to find an actual factor. Finding a number that shares a factor with N is also sufficient due to Euclid's algorithm, as it can be used to find the greatest common divisor (GCD) between N and g. A function that calculates the GCD of two numbers x and x is denoted to as $gcd(x, y)$. If g and N share a common factor, this algorithm can be used to find p or q. In any other case, a period r of the function $f(g) = a^g mod\,(N)$ needs to be found, so that:

$$a\ mod\,(N) = a^r mod\,(N)$$

If $r\ mod\ 2 = 1$ a different g needs to be chosen and the algorithm is restarted.
Otherwise calculate the following:

$$x = gcd\left(g^{\frac{r}{2}} + 1, N\right)$$

If $x = N$, a different g is chosen and the algorithm is restarted. If $x < N$, then x is a nontrivial factor of N. The other factor can then be calculated as $y = gcd\left(g^{\frac{r}{2}} - 1, N\right)$. [Sch19]

While the whole algorithm could also be executed on a normal computer, the determination of r would not be computationally feasible. This is the section of Shor's algorithm that requires a QC in order to run efficiently. The exact number of qubits required to successfully execute this algorithm in practice is not clear at the moment, as Gidney et al. [GE19] conclude. In theory, the algorithm as described by Shor requires $3n$ qubits. Beauregard [Bea03] described a version which reduces the required number to $2n + 3$ qubits. In practice however, qubits require some form of error correction. Depending on the method used, this increases the number of required qubits considerably. Gidney et al. estimate that around 20 million qubits are required to factor RSA-2048. A QC of these dimensions does not exist at the moment.

Discrete Logarithm Problem

The DLP can be calculated in a similar way to the factorization problem. While Shor's original paper already proposed a way to solve the DLP, a number of improvements have

been made in the subsequent years. Martin Ekerå [Eke16] presented a modification of Shor's algorithm that is able solve the DLP with a success probability of 60 to 70%. Proos et al. [PZ03] showed that an attack on 160bit ECDSA would be possible using 1000 qubits, half the number required for the factorization of the security-wise equivalent RSA-1024.

2.4.2 Grover

Another possible application of large scale QC was described by Grover in 1996[Gro96]. He introduced a method for searching in random databases. Given a database with N unsorted entries an the task to find element w inside it, a classical computer would need to check $\frac{N}{2}$ entries on average. Using Grovers algorithm on a QC, only $\sqrt{N}$ steps are necessary to find the element.

Grove's algorithm can also be used to find collisions within the codomain of a hash function, as this can also be considered an unsorted database. Brassard et al.[BHT98] described a quantum birthday attack on cryptographic hash algorithms. Amy et al.[AMG$^+$16] estimated the requirements for a QC and the computational costs of using Grover's algorithm for attacking SHA-2 and SHA-3, both with an output length of 256 bits. They estimate a number of $2^{12,6}$ and 2^{20} qubits and a cost of $2^{153.8}$ and $2^{146.5}$ surface code cycles respectively. A Surface code cycle is defined as a operation that involves some QC steps and a considerable amount of classical processing. Amy et al. assume the temporal cost of one surface code cycle and one hash function invocation to be equal. Assuming a CPU has a clock speed of 4GHz and requires ≈1000 cycles per hash function, it would be able to calculate ≈ 4 million hash functions per second and core. This rough estimation would lead to a runtime of $4,97 * 10^{39}$ seconds per core and therefore would not be computationally feasible even if a large enough QC existed. At the moment, cryptographic hash functions are considered to be quantum-proof and are therefore building blocks for post-quantum cryptography.

2.5 Hash functions

In general, a hash functions $H : \{0,1\}^* \rightarrow \{0,1\}^n$ maps input data of arbitrary length into output of a fixed size. Even small changes to the input data must result in completely different outputs. To be considered a cryptographically secure hash function, is has to fulfill a number of properties [Sma15]:

- Preimage resistance:
  For a given hash value y it should be computationally infeasible to find an input m such that $H(m) = y$.

- Second preimage resistance:
  For a given input m is should be computationally infeasible to find an $m' \neq m$ with $H(m') = H(m)$.

- Collision resistance:
  Is should be computationally infeasible to find two inputs m and n, $m \neq n$ with $H(m) = H(n)$.

Collision resistance is the strongest security assumption. It also impies second preimage resistance. Likewise, the second preimage resistance also impies preimage resistance [KL07].

The security of any given hash function is related to its key length n. Finding a preimage or a second preimage by random requires 2^n tries on average. Finding a collision only requires $2^{n/2}$ random executions due to the birthday problem [AM96].

2.6 Hash-based signatures

In order to provide security even if sufficiently powerful quantum computers are available, cryptographic functions with quantum resistant mathematical problems need to be established. Hash functions are believed to be quantum secure. Given a cryptographically secure hash function with large enough output, there is no known way, be it using a classical computer or a quantum computer using Grovers algorithm, to sufficiently perform attacks. Apart from hash based signatures there are a number of other mathematical problems that can be used for post-quantum cryptography. In the final round of the standardization process for post-quantum signature schemes by NIST [3], the finalists CRYSTALS-Dilithium [DKL+18] and FALCON [PAF19] are lattice-based while RAINBOW [DS05] is based on multivariate cryptography. A common trait among all those post-quantum algorithms is the significantly higher cost of signing operations and larger signature sizes compared to conventional signature algorithms. An advantage of hash-based algorithms is the fact that these are well understood. Some signature schemes even allow the usage of only second preimage resistant hash functions by applying pseudorandomly generated bitmasks to their input [HBG+18]. The overall security of the scheme used relies solely on the used hash function. If it gets vulnerable to attacks in the future, it can be replaced with another hash function while the overall scheme does not need to be modified further.

2.6.1 Winternitz One-Time Signature Scheme

The Winternitz One-Time Signature Scheme (WOTS) is one of those schemes that only relies on the security of the hash function used. Being a one-time scheme, each key pair can only be used to sign a single message. An extension to his scheme are the Merkle Signature Scheme (MSS) and the eXtended Merkle Signature Scheme (XMSS). They allow to sign multiple messages using one *pk* and multiple associated *sk*. They are not limited to one tree but can also be extended to have multiple layers of tree. The following section gives a description of WOTS+, a variant of WOTS introduced by Hülsing et al. [Hül13]. Afterwards, XMSS and its multi-tree variant XMSS^{MT} will be described.

WOTS+ Parameters, functions and addressing scheme

Before the actual WOTS+ process will be described, the important parameters used alongside a description will be explained here. The parameter $n \in \mathbb{N}$ denotes the length of the message as well as a single element of the private key, public key and signature element respectively in bytes. It is common practice to chose the value of n to be of the same as the number of bytes output by the underlying hash function. This has two reasons. First, n is a security parameter and has therefore be chosen to be of a sufficient size. This estimation of what is a reasonable size is already made by the choice of the hash function. Secondly, using the same size for both cases simplifies the implementation. The Winternitz parameter $w \in \mathbb{N}$ is used

[3] https://www.nist.gov/news-events/news/2020/07/pqc-standardization-process-third-round-candidate-announcement, visited 27th of July

Addressing scheme			
size	OTS Hash	L-tree	Hash tree
32 bits	layer address	layer address	layer address
64 bits	tree address	tree address	tree address
32 bits	type = 0	type = 1	type = 2
32 bits	OTS address	L-tree address	0-Padding
32 bits	chain address	tree height	tree height
32 bits	hash address	tree index	tree index
32 bits	keyAndMask	keyAndMask	keyAndMask

Table 2.1: Addressing scheme

to reach a trade-off between signature length and signing time, where larger values generate shorter signatures but require more computation time. A value of 4 or 16 is recommended as those values facilitate a easier implementation while also providing an optimal time-space trade-off [Hü13]. The parameter len is equal to the number of elements in the public key pk, private key sk and signature, where each element is an n-byte string. len itself consists of two substrings len_1 and len_2 where

$$len_1 = ceil(8n/log_2(w)$$

$$len_2 = \frac{floor(log_2(len_1 * (w-1))}{log_2(w))} + 1$$

Table 2.2 shows the different kinds of functions used in the signature scheme described in the following section. The keyed cryptographic hash functions F, H and H_{msg} all take different input but can all be implemented using the same hash function underneath. For each hash function, a key is used alongside the actual input string to randomize the call. This key is generated using the pseudorandom function PRF.

The address used as an input for PRF points to specific WOTS+ key pairs or to nodes within the XMSS tree structure. Table 2.1 shows this addressing scheme. Layer and tree address are used in the multi-tree variant of XMSS, referring to the layer of the tree within the multi-tree and its index on that layer respectively. The field type is used to differentiate between the three kinds of signature. OTS address and L-tree address refer to the index within the leaf nodes of the next higher order hash tree. In Hash tree addresses, this field is padded with 0. Chain address is the position within the chaining function while hash address refers to the index within the len segments of a WOTS+ key pair. In the case of the L-tree and hash tree addresses those two fields indicate the height and index of the node being input for the next computation. The last field, keyAndMask is set according to whether PRF is used to calculate a hash function key or a bitmask.

WOTS+ Key Generation

The public key pk is calculated from chained hash function calls on all len key elements. This chaining function requires a keyed cryptographic hash function F that accepts two byte arrays of length n and a 32-bit index as inputs and outputs a hash value of length n. The actual value of n is often dependent on the output of the underlying hash function used in

Function modules			
function	type	input	output
F	hash function	n-byte key n-byte string	n-byte string
H	hash function	n-byte key 2n-byte string	n-byte string
H_{msg}	hash function	3n-byte key arbitrary string	n-byte string
PRF	pseudorandom function	n-byte key 32byte address	n-byte string

Table 2.2: Function overview

F. The initial input of the chaining function is a private key. Each private key is an array with length len with each element being a uniformly random byte-String of length n. Its correspondent public key features the same characteristics and is calculated by chaining hash function calls. The input to this chained hash function is a single element of the private key array. First a bitmask is calculated using PRF and applied to this element. This step reduces the security requirement of the used hash function from collision resistance to second preimage resistance. Secondly, the resulting output is used as the input of the next chain. These two steps are repeated $w - 1$ times, producing $w - 1$ intermediate results in the form of the hash function outputs. The result of the last iteration of this chain is the public key array element of its respective private key element. This chaining function is applied to all len key parts, resulting in a public key consisting of len n-byte arrays.

WOTS+ Signature Generation

The message to be signed is split into len_1 parts of equal length. Each of these parts is then mapped to integers between 0 and $w - 1$. Next a checksum of len_2 parts is constructed in a similar way. These again are mapped like previously and appended to the first array, resulting in a len length array. Depending on the value i of the individual elements, the $w - i$th intermediate result of the chaining function on each respective element of the private key is selected. By appending these nodes, the signature itself is created.

WOTS+ Signature Verification

To verify a signature sig, the receiver again calculates an array of integers like the signer did during signature generation. Depending on the value i of the individual calculated elements, the respective elements of the signature are hashed further i times, thus being hashed a total of $w-1$ times and therefore being equal to the segments of the public key. The concatenation of the result of these hash chains is compared to the provided public key. The verification is successful if they are equal.

2.6.2 Forward secrecy

Forward secrecy in general is the property of encryption and signature algorithms that ensures that encrypted communication or issued signatures remain secure or valid even if the sk gets

compromised in the future. They way this property is achieved depends on the algorithm. In the context of OTS schemes this means that an attacker who gets hold of the global sk at the time the i-th OTS sk_i was active must not be able to recreate sk_{i-1} and any of its predecessors. In XMSS this is achieved using a trapdoor function in the form of a pseudorandom generator that updates the seed required for computing the OTS sk and overwrites the current one in the process. There is no way to revert this process, thus making is impossible to reconstruct a previous sk. Braun et al.[BKH14] also showed that forward secure signature schemes can be used to maintain the property of non-repudiation. In this case a key revocation does not invalidate this property as previously used sk remain valid.

2.6.3 XMSS - Extended Merkle Signature Scheme

Using the WOTS+ scheme described above, creating quantum resistant signatures is already possible. However, the problem lies within the key management. For every signature created, one pk is consumed. This is problematic as every signature recipient would also need to obtain an individual pk. This is where XMSS comes into play. Instead of verifying a signature created by some sk against its respective pk, multiple OTS signatures can be verified against a single XMSS pk. XMSS, introduced by Buchmann et al. in 2011 [BDH11], can be considered an extension to WOTS+. It organizes WOTS+ pk in a binary hash tree. The individual pk represent the leaves of the tree with one single global pk as the tree's root. Based on the Merkle Signature Scheme (MSS) introduced by Ralph Merkle [Mer90], XMSS also provides forward secrecy. The following section describes the structure of XMSS starting at the base with the already established WOTS+ public keys. XMSS is constructed in a bottom up manner, so the individual components will also be introduced in this order.

2.6.4 L-tree construction

As mentioned, XMSS hash tree construction starts at the bottom layer with WOTS+ public keys. The len key elements, each with a length of n bytes, are compressed into a single n-byte array using a binary hash tree called L-tree. This tree has len leaf nodes, each holding a single element of the WOTS+ pk and the compressed result in the root. An L-tree root node is one leaf node of the XMSS hash tree. The rest of this section describes the construction of this structure. A visualization of the process is given in Figure 2.1.

The starting point are the len segments of the key where each can be considered a leaf node of a binary hash tree. Two adjoining nodes are combined into their parent node in the following way: Using a public seed and the address of the new parent node as an input for three PRF calls, two bitmasks and a key are created. The bitmasks are applied to the child nodes. The resulting strings are used as an input to the hash function H alongside the calculated key. The result of H is the value of this parent node. This routine is applied to to all siblings of each layer. If there is an odd number of nodes on any given tree level, the remaining node is lifted up to the next layer. This is repeated until a single root value of n bytes is computed.

2.6.5 XMSS Hash Tree construction

The XMSS hash tree is a perfect binary hash tree with every node and leaf containing an n-byte array. The tree construction itself is almost identical to the L-tree construction. The tree is constructed from the bottom layer containing 2^h L-tree root nodes. To construct

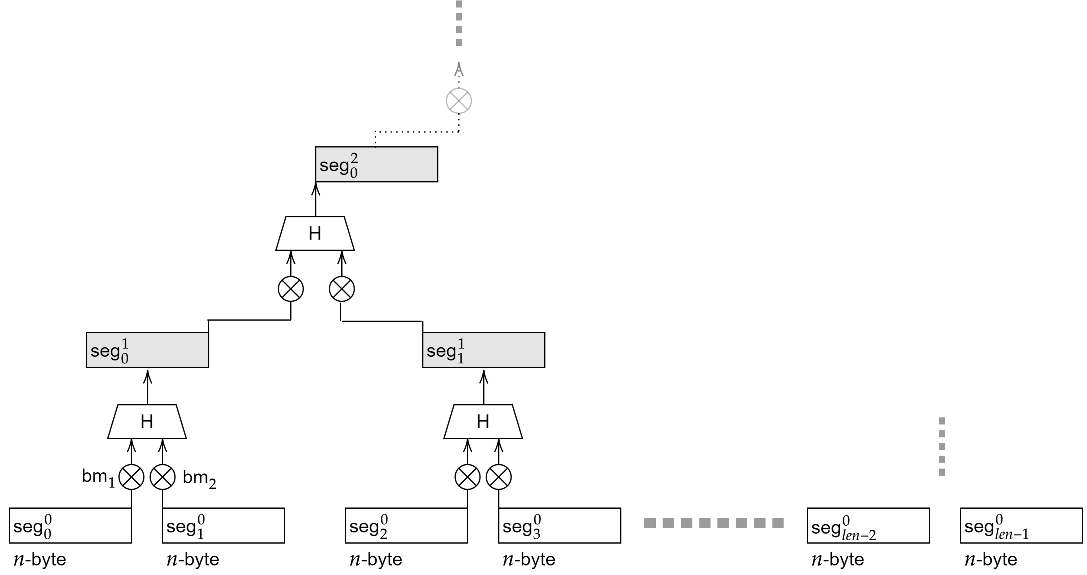


Figure 2.1: L-tree construction visualized: Segments of the uncompressed WOTS+ pk and intermediate results are denoted as seg_x^y with x being the position on that L-tree layer and y being the layer within the tree

a parent node on layer $h + 1$, bitmasks and a key are computed using PRF. The masked children of level h are again hashed using H with the respective key to obtain the value of the new parent node. The masking step is necessary in order to reduce the security requirement of the keyed hash function from collision resistance to second preimage resistance. This routine is again applied on all sibling nodes of any layer, halving the number of nodes per layer with every height increment until the root of the tree is reached. The value of the root node is the global public key of XMSS.

2.6.6 XMSS signature generation

The actual signing of a message is similar to the routine described in Section 2.6.1 with the difference being the use of H_{msg} to hash the input message. H_{msg} receives a key consisting of the current leaf index, the pk and a pseudorandom value R generated by PRF. The message is then hashed to a n-byte value which is signed the same way as described in Section 2.6.1. The resulting array of $len * n$ bytes is the WOTS+ signature. In addition, the so called authentication path of $h * n$ bytes is appended to the signature. Consider a path from the hash tree node used for the current signature all the way to the root node. For each node that lies on this path, the authentication path contains its sibling on the respective level. Figure 2.2 illustrates the authentication path.

2.6.7 XMSS signature verification

Alongside the actual message, the receiver gets an XMSS signature with a structure as shown in Figure 2.3. The first step of the verification process is similar to the one described in section 2.6.1. The difference in XMSS is the missing WOTS+ public key that would be used

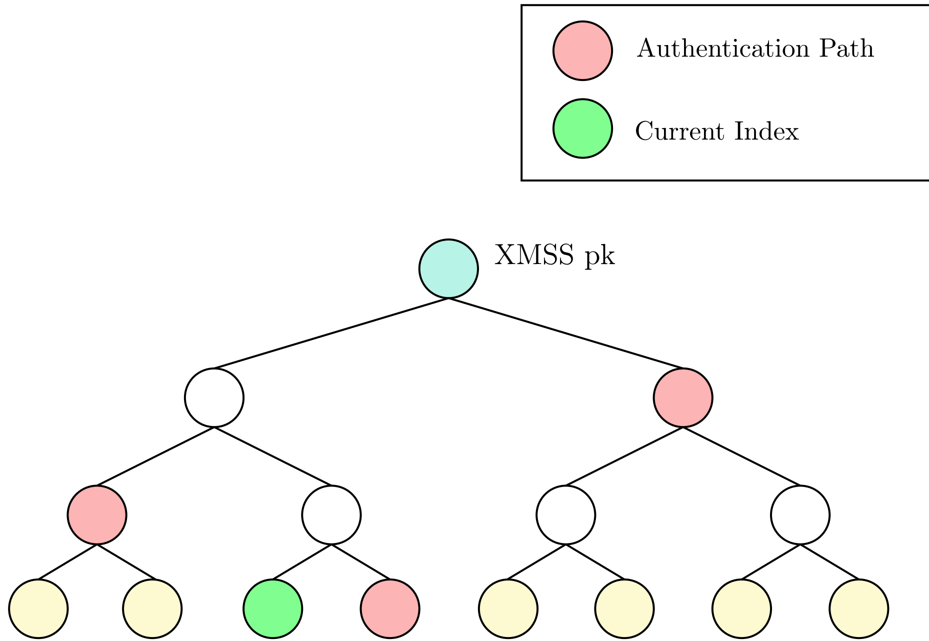


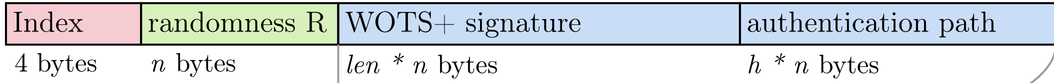
Figure 2.2: Authentication Path visualized

for comparison. The computed pk is then compressed using the L-tree technique described in Subsection 2.6.4. The seed required for PRF input is part of the signature in the form of the randomness R. In the final step, the verifier needs to construct the XMSS hash tree. To this end, the authentication path is used. Its first n-bytes represent the sibling of the calculated L-tree root within the XMSS hash tree. They are used to calculate their parent on the next higher tree level, whose sibling is again part of the authentication path. Iterating over the tree levels, repeatedly combining the calculated node with its respective sibling stored in the authentication path, the root of the hash tree is eventually calculated. If its value is equal to the publicly known global pk, the signature is valid. The integrity of the message is ensured through the correctness of the constructed WOTS+ pk which is in turn required to construct the global pk, which is used to check the authenticity.

2.6.8 XMSS$^{\text{MT}}$ - Multi-tree XMSS

As each XMSS private key can only be used once but the public key should be valid for a long time, the tree needs to hold a sufficiently large number of leaves. However constructing such a tree is computationally expensive. Consider a tree with a height $h = 20$. This tree is able to hold 1.048.576 WOTS+ key pairs. In order to be able to compute the global public key, all WOTS+ public keys must first be constructed. This alone takes $2^{20} * len * (w - 1)$ calls to H and PRF each. This is infeasible to calculate on lower performance devices such as SmartCards. Multi-tree XMSS (XMSS$^{\text{MT}}$) [HRB13] is a technique that allows those numbers of key pairs while only requiring a fraction of the pk computations. In order to hold 2^h signatures, not a single tree of height h is used but d layers of trees, each with a height of h/d. This multi-tree has one tree on level $d - 1$ with an equally sized tree under every leaf. Consequently the bottom most layer contains $2^{h-h/d}$ trees with a combined number of 2^h leaves. The trees on the bottom layer hold the OTS public keys and are used to sign the

XMSS



XMSS^{MT}

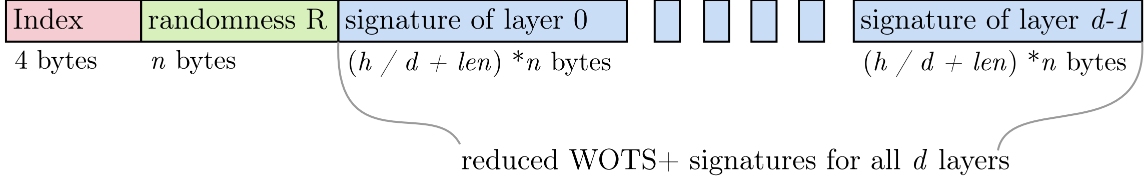


Figure 2.3: Signature structures of XMSS and XMSS^{MT}

actual messages. The trees on the next higher level are used for signing the root node of the respective tree below. In order to generate the global *pk* and the authentication path for the first $2^{h/d}$ key pairs, only the first subtree of every level needs to be constructed. Those subtrees are referred to as active trees in the following sections. Whenever all nodes on a active tree are consumed, the next tree on the respective layer becomes the active tree.

XMSS^{MT} Signature structure and verification process

The signature verification process is similar to the single tree XMSS. The recipient receives a signature with the structure shown in Section 2.3. The header is the same as for XMSS with the index and the randomness R, followed by a set of WOTS+ signature and an authentication path for each of the d subtree levels. The root of each subtree is reconstructed by first calculating the WOTS+ *pk* using the signature and then using the resulting values together with the nodes of the authentication path to construct the tree. The resulting root node is then again handled like a message that is verified against the subtree of the next higher level. This is repeated up until the top layer of the tree, where the calculated root is compared to the known global *pk*.

2.7 Tree traversal techniques

Treehash is a function that calculates the value of any given node within the XMSS hash tree. The public key construction described in Section 2.6.5 is essentially an execution of the treehash algorithm with the hash tree root as the input. The same routine can also be executed with any other node. For leaf nodes, the result is equal to the construction of a single compressed WOTS+ key. Treehash can be very expensive, as for a node on height v, the construction of 2^v WOTS+ keys and $2^v - 1$ nodes is necessary. Improvement of the algorithm itself is not possible. The efficiency of treehash is already optimal, as the construction of any given node requires all the aforementioned parts.

When building the authentication path without any optimization, 2^{h-1} treehash executions are required with every treehash instance depending on unique keys. As a result, in

the process of the whole authentication path computation, $2^h - 1$ compressed OTS keys and $2^h - 1 - h$ nodes would have to be calculated, which makes this almost as costly as the *pk* generation. In this section, some possible improvements to the authentication path computation will be described.

2.7.1 Notation and general workflow

The descriptions and nomenclature of the following section is based on the paper that introduced the BDS tree traversal algorithm by Buchmann et al.[BDS08]. Nodes are referred to as $y_v[j]$, where $v = 0, ..., h$ is the height of the node within the tree and $j = 0, ..., 2^{h-v} - 1$ is the index of the node on layer v. The index of the current leaf is referred to as $\varphi \in \{0, ..., 2^H - 1\}$. $STACK$ can be considered an stack of tree nodes with operations push and pop. The algorithm calculates every leaf underneath the input in consecutive order and pushes them to $STACK$. Every time two nodes of the same height are on top of $STACK$, they are combined to their parent node.

Additional BDS notation

In addition to these notations, some additional descriptions are required for BDS, an algorithm that will be described later in this section. $AUTH$ is the array used for saving the current authentication path with $AUTH_v$ referring to the node on level v. The same notation is also used for $TREEHASH_v$, denoting the treehash instance on layer v. The value τ is defined as the height of the first node on the path from leaf φ to the root of the tree that is a left node. If φ is a left node itself, τ is 0. From τ of the leaf φ, the tree level of the nodes required for the authentication path of leaf $\varphi + 1$ can be derived. A new right node is required on levels $v = 0, .., \tau - 1$ and a new left node on height τ. In order to speed up the computation of left nodes like described in section 2.7.2, the array $KEEP_v$ is introduced, where $v = 0, .., h-2$. Additionally, a single node RETAIN, the rightmost node on level $h-2$, is stored in memory. This is a memory efficient optimization, as RETAIN is the only new node required on this level.

2.7.2 Basic optimization strategies

During the construction of the global public key all underlying nodes are already computed. One simple approach to minimize the time required for the path construction would be to store all nodes in memory. However this is not possible for bigger tree sizes, especially when memory is a critical resource, like it is the case with SmartCards. For example, a tree of height $h = 20$ with $n = 32$ requires $2^{20} * 32$ bytes = 32MB of memory, which exceeds the capacity of any SmartCard available.

Storing previous path in memory

The storage space available has to be used strategically. One fact that can be exploited is that a node on level v is required in 2^v consecutive authentication paths. By storing the first authentication path in memory and only calculating the value of nodes that are different from the previous signature. In most cases a majority of the path can be reused with the most costly nodes on the highest layer requiring the least frequent recalculations. This improvement also only needs $h * n$ bytes of storage which amounts to 640 bytes in the

previous example. The problem with this example is the worst-case signing time. After 2^{h-1}, or half the total signatures, the whole authentication path needs to be updated. As a result, the worst-case-cost is still $2^h - 1$ leaf and $2^h - 1 - h$ node computations.

Storing right child nodes

Another observation that can be made is the fact that if a new left node is required for the authentication path, its children have already been part of previous paths. More specifically, if a new left node on level v is required for the authentication path of signature i, its left child was part of the previous one and its right child was used in signature $\left(i - 1 - 2^{v-1}\right)$. If this new node needs to be calculated, the left child is therefore still in storage from the previous round and by additionally storing all right child nodes, only a single node computation is required. This also only requires an additional storage of $\left(\lceil \frac{h}{2} \rceil * n\right)$ bytes.

Storing the top layers in memory

Right child nodes cannot be computed in a similar optimized way as left ones. Right children of the highest levels are therefore the most costly to compute. However, as every node is computed during *pk* construction, those can be stored in memory, avoiding their construction during a signing operation and thus reducing the average signature generation time. The parameter k is therefore introduced, offering a trade-off between time and memory by storing all right child nodes of the top k layers during *pk* construction. For this, additional memory of $2^k - 2 * n$ bytes is required. While this reduces the worst-case signing time to $2^{h-k} - 1$ leaf and $2^{h-k} - 1 - h - k$ node computations, this can still be problematic. Depending on the memory available, the worst-case may still differ significantly from the average time. For any digital signature algorithm this is an undesired property.

Distributing computations

For every second signature, only one node of the authentication path has to be changed. Those signatures would therefore have a significantly lower runtime than the average. This time difference can be used to calculate nodes ahead of time, reducing the worst-case runtime down to the average. To facilitate this, a separate treehash instance per level is created. The treehash algorithm is extended by an update routine, with each update calculating one new hash tree leaf and pushing it to $STACK$. The stored nodes are then combined as far as possible. The computation of the next right node on level v is started as soon as the current one on this level becomes part of the authentication path.

2.8 BDS Tree traversal

BDS tree traversal was first introduced by Buchmann, Dahmen and Schneider in 2008 [BDS08]. By combining all the optimization techniques mentioned in Subsection 2.7.2, it achieves a worst-case runtime of $(h - k)/2 + 1$ leaf and $3(h - k - 1)/2 + 1$ node computations. The following section describes the way the above mentioned optimizations are combined.

2.8.1 BDS Scheduler

In order to make sure that every right node is fully constructed by the time it is needed in an authentication path, scheduling its respective leaf computations is necessary. While there are approaches to also schedule node computation [Szy04], for BDS only leaves are considered as node computations from existing children are significantly less expensive. The way new nodes are computed depends on whether they are left or right nodes. The construction of the authentication path for signature φ is performed directly after the signature for φ is created.

Left nodes

For left nodes, the array KEEP is used to store certain right authentication nodes. Whenever the path construction for leaf φ is finished, $\tau \neq 0$ and $AUTH_\tau$ was updated and now contains a right node, $AUTH_\tau$ is stored in $KEEP_\tau$. This node is required in round $\varphi' = \varphi + 2^\tau$, when it is used for the construction of a new left node on level $\tau + 1$ together with a left node in $AUTH_\tau$.
Whenever $\tau = 0$ for a leaf φ, the single new left node required for the authentication path of $\varphi + 1$ is the current leaf node which is directly calculated.

Right nodes

As previously mentioned, right nodes cannot be constructed using child nodes of previous authentication paths, none of their children were computed before. This is where the distributed calculation over multiple signing cycles comes into play. During every signature generation, The signature $\varphi + 1$ requires new right nodes on the authentication path on levels $v = 0, ..., \tau - 1$. The right child node on height $h - 2$ forms an exception, as it is already stored in RETAIN and does not need to be computed. After each signature generation, $(h - k)/2$ treehash updates are performed. An update is always performed on the treehash instance with the minimum height node on its stack. This number of updates is sufficient for finishing every node computation in time. On average $(h - k + 1)/2$ leaves and $(h - k - 1)/2$ are calculated during every BDS update.

2.8.2 Cost of subtree switches

With every additional subtree level within an XMSS$^{\mathrm{MT}}$ structure a new BDS instance is running, managing the authentication path computation on its respective layer. For most signature operations this is not problematic, as changes in the authentication path only happen on the lowest tree layer. However, after h/d signatures, its leaf nodes are consumed and a new tree is required for the next h/d signatures. This is where a BDS update is performed on the second layer from the bottom, as a new signature is required to authenticate the root value of the new subtree on the bottom layer. This requires a signing operation and an update of the authentication path. Now a XMSS$^{\mathrm{MT}}$ tree can consist of more than two layers. This means that every $2^{(h/d)*y}$ signatures, with $0 < y \leq d - 1$ being the respective subtree level, a switch on this level is required. This also implies that if a switch is required on a layer $y > 1$, the same is also true for all layers $< y$. An XMSS$^{\mathrm{MT}}$ tree consisting of d subtrees requires up to $d - 1$ signature calculations and BDS updates for a single signing operation.

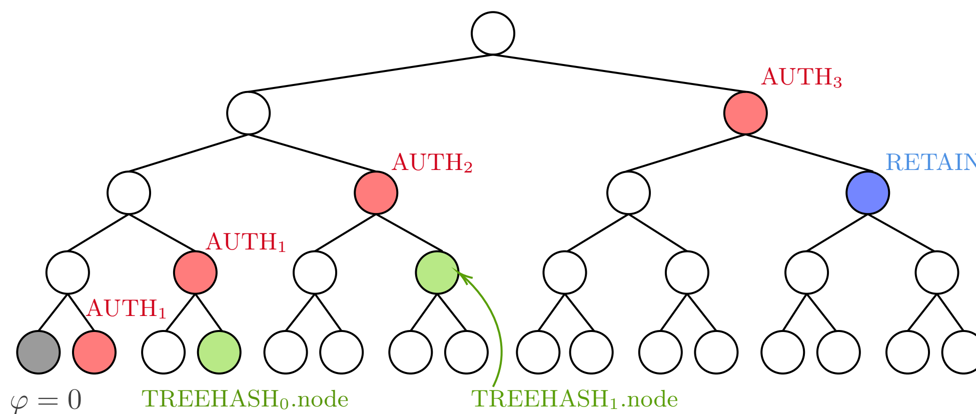


Figure 2.4: Initialization of the BDS algorithm. [BDS08]

2.9 SmartCards

Before describing the individual components of SmartCards (SC), a definition of what differentiates a SmartCard from a standard chip card is given in the following section. Both types of cards share a common physical appearance, being of equal size and containing a similar looking contact plate with the logic board located underneath. Up until around 2002, the most commonly used types of integrated circuit (IC) cards were memory cards. Being used as higher security alternatives to magnetic stripe storage cards, they can hold 1-4KB of data. This data is stored on non-volatile memory and can only be manipulated by the card reader directly. In contrast, IC Microprocessor cards, in the following referred to as SmartCards, contain a processor that is capable of performing complex calculations and directly accessing and modifying the on-chip memory[Ort03, BP]

2.9.1 Internal Components

Figure 2.5 shows the hardware components of a standard SmartCard. The SmartCard's CPU is able to directly access both RAM and EEPROM. The actual internal components of any specific SC differ depending on the actual model. Some cards contain separate cryptographic coprocessors for hardware accelerated execution of algorithms such as RSA, AES and DES while others rely on software implementations of those functionalities. Where this coprocessor is present, it is able to directly access RAM and communicate with the main CPU.

Processor

The CPU on a SmartCard is typically an 8, 16 or 32-bit processor running at a clock speed of 3.7 - 10.0 MHz. Every time a SC is inserted into a reader or a reset is performed, the card sends an answer to reset (ATR) dataframe constaining information about hardware requirements for running the card. These include the maximum clock rate, required operation voltages and current and optional parameters depending on the card. The clock itself is provided by the terminal through a special contact pad. Modern SmartCard processors often offer an orthogonal instruction set, allowing them to execute instructions for all the above mentioned address lengths.

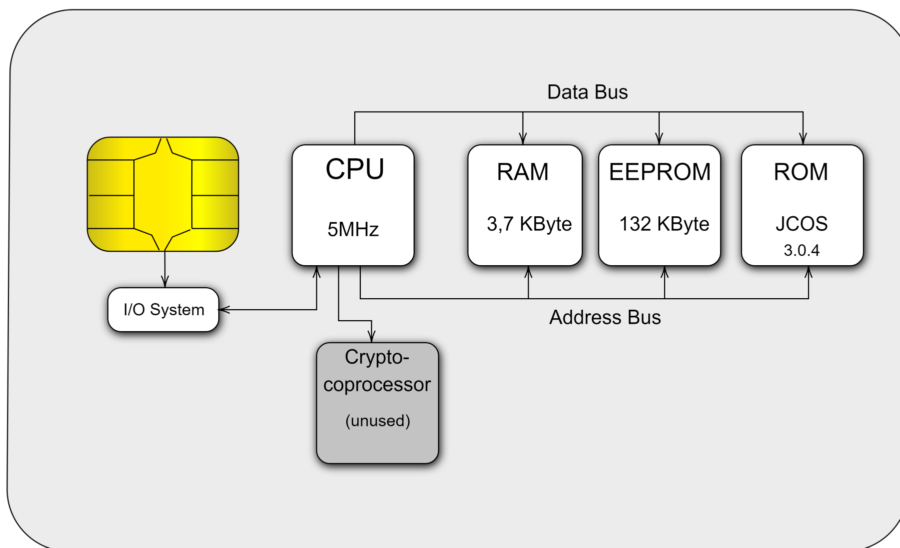


Figure 2.5: Typical SmartCard architecture

Memory

A SmartCard contains three types of memory. The read-only memory (ROM) contains the SC's operating system and is not accessible from a programmer's perspective. It is written to using masked programming during the cards production. The two types of memory available for data storage and temporary variables are a volatile RAM and a non-volatile memory in the form of either EEPROM or flash memory. For many years, EEPROM was the standard technology used, but flash storage is getting more common in use cases where a larger storage capacity is required, as it offers higher storage density. While both technologies use basically identical memory cells, they are structured differently. EEPROM allows bytewise read and write access while flash memory can only be accessed blockwise which negatively impacts performance. Typical EEPROM sizes vary between 8 and 245 KB, while some flash cards offer storage capacities of several Megabytes. The RAM capacity of SCs range from 256 bytes to 12 KB depending on the model and the intended use cases. Accessing non-volatile storage is significantly more expensive than accessing objects in RAM with write times being slower by a factor of more than 30.000 [RE10].

2.9.2 SmartCard Interface

The general communication between a PC and a SmartCard is defined by the PC/SC specification. While other standards exist , for example CT-API mainly on the German market, the international de-facto standard is PC/SC. It ensures a common communication standard over different operating systems, SmartCard readers and SmartCards. An official implementation of PC/SC is included in Microsoft Windows since Windows 2000, while a free version called PC/SC Lite is available for Linux and Unix-based systems. PC/SC defines a layered architecture that splits the communication chain from the card on one end to the software using the card on the other end. An Interface Device (IFD) Handler hides the functionalities and complexities of the SmartCard reader and its lower-level communication with the card itself from user applications by providing a common API. Figure 2.6 shows the layerd archi-

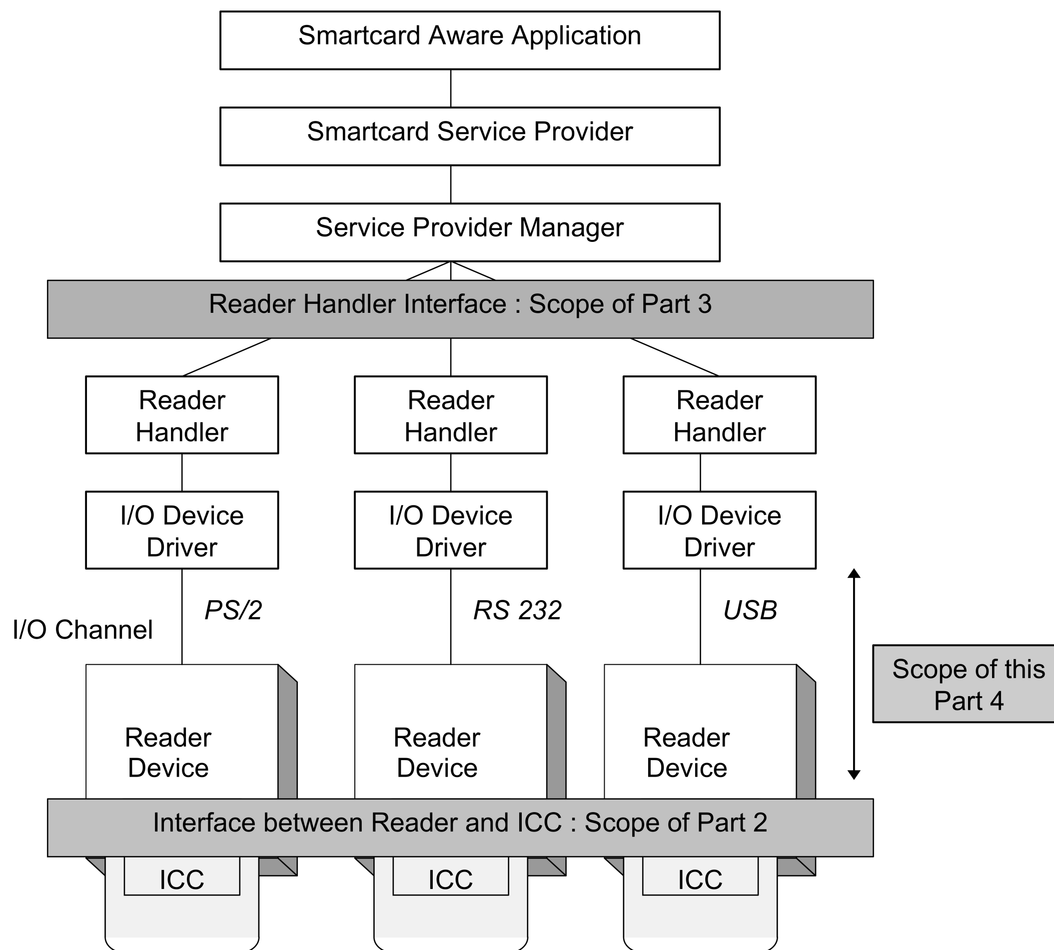


Figure 2.6: IFD-Subsystem as shown in [IFD05]

tecture. The Service Provider Manager handles all IFDs present in the system. It detects insertion and removal of cards, handles assingments of cards to their respective applications and provides an API for communicating with the SmartCard. A higher level API that can be used by the SmartCard aware application is provided by the SmartCard Service Provider. This software library acts as a translation layer between common higher-level API functions provided to the application and card-specific commands in the form of Application Protocol Data Units (APDUs).

2.9.3 APDU - Application Protocol Data Unit

Data between the host application and the SmartCard is exchanged using Application Protocol Data Units(APDUs). In general, an APDU is a data block of variable length. Its structure is defined in ISO 7816-4 [ISO20]. Command APDUs are used to send instructions to a SmartCard while response APDUs are sent from the Card to the receiving function over the PC/SC interface.

Command APDU Structure			
Type	Name	Length	Description
CLA	Class	1 byte	Class of this command
INS	Instruction	1 byte	Instruction identifier
P1	Parameter 1	1 byte	First Parameter for instruction
P2	Parameter 2	1 byte	Second Parameter for instruction
Lc	Length command	0-3 bytes	Length of the Data field
Data	Data	Lc bytes	Data payload
Le	Length expected	0-3 bytes	Length of the expected response data

Table 2.3: Command APDU Structure

Class	**Header**				**Optional**		
1	CLA	INS	P1	P2			
2	CLA	INS	P1	P2	Le		
3	CLA	INS	P1	P2	Lc	Data	
4	CLA	INS	P1	P2	Lc	Data	Le

Table 2.4: Command APDU classes

Command APDUs

Command APDUs are categorized into four cases depending on their actual structure. In any case the header needs to be present. Case 1 APDUs are used for simple commands where neither data is required nor a response is expected. Case 2 and 3 are used for cases where either only data is sent or received respectively while in case 4 payload is sent and expected to be received. Table 2.4 shows their structure. As can be seen in Table 2.3, the fields Lc and Le can be of a length between 0 and 3 bytes. In a basic case, the length of those fields is 1 byte, therefore being able to indicate a data and response length of up to 256 bytes. This is a constraint if larger inputs and outputs are required, as is the case especially with hash-based signatures. Extended APDUs where introduced to enable data bodies of a length of up to 65635 bytes by providing 2 bytes for length encoding. In this case, the leading Lc and Le bytes are encoded as 0x00, indicating an extended APDU, with the actual length encoded in the two tailing bytes. The JavaCard OS supports extended APDUs since version 2.2.2 with the constraint of only supporting lengths of up to 32.767, as its API defines the length input as a signed short [Cor].

Response APDUs

For every command APDU, one response APDU is sent back to the calling software. Depending on the number encoded in the command APDU's Le field, the response's body holds

Response APDU Structure			
Type	Name	Length	Description
Data	Body	le Bytes	Response data of length given in CMD-APDU
SW1	Status Word 1	1 Byte	response byte 1
SW2	Status Word 2	1 byte	response byte 2

the defined amount of bytes as payload but can also be absent for case 1 and case 3 command APDUs. The status words SW1 and SW2 are present in any case, indicating if the command was run successfully or an error occurred. If more data needs to be returned than fits in a single APDU, it has to be actively fetched by the calling software using additional command APDUs, as SmartCards never initiate data transfer but only react to incoming commands.

2.9.4 Javacard Operating System

The operating system of a SmartCard provides many basic functionalities. It is responsible for controlling user authentication, for example via a PIN, managing memory access and handling I/O between the physical contact pads or a wireless receiver and the currently running application on the SmartCard.

JavaCard OS (JCOS) is one of those operating systems. The JavaCard programming language is a subset of standard Java. This comes especially apparent in the applet build process. JavaCard code is first compiled into a class file using a standard Java compiler and then converted and linked into a CAP file, standing for converted applet, by a JavaCard converter [Che00]. This converter performs tasks that would be performed by the Java VM on standard Java during class-loading time. These tasks include checks for JavaCard language subset violations and bytecode optimization. CAP files utilize the JAR file format with a minimum size optimization. They can then be uploaded and installed on a JavaCard as an applet. A single card is capable of storing multiple applets simultaneously. Every time one applet is selected for execution, the previously active applet is halted with its state being stored in persistent memory for later resumption. The individual applets are isolated from another. The following sections provide an overview over JavaCard specific characteristics. As of the time of writing, the latest version of JCOS is 3.1. While this version was released by Oracle in May 2019, no card supporting this version is available yet. Card manufacturers are only slowly adopting new standards with JCOS 3.0.4, released in August 2011, being the latest version with cards available.

Javacard VM

On standard PCs, the Java Virtual Machine runs as an operating system process, with all its applications and their objects being automatically destroyed as soon as the process is terminated. The JavaCard Virtual Machine (JCVM) works differently in this regard. After its initialization, the JCVM's lifetime does not end until the card is destroyed. When power is removed from the card, the VM execution is paused and resumed as soon as the card receives power again. This is achieved by storing all heap objects on the persistent EEPROM. The required memory is permanently reserved for this object and cannot be freed again. As a result of this architecture, a garbage collector is not present on the JavaCard runtime environment and all objects need to be reused in each execution cycle. This also means that non-primitive data types such as arrays must only be created during the installation of an applet or in its initialization phase. Accessing objects stored on the non volatile EEPROM is significantly more expensive than accessing objects in RAM. This negative performance impact can be avoided by explicitly creating non-primitive objects in RAM. In contrast, primitive data types can be created during the applet execution, as those are exclusively stored on the card's RAM.

2.9.5 JavaCard limitations

As stated in Subsection 2.9.4, the JavaCard programming language is a subset of standard Java. Many complex data types and features with higher system requirements are unsupported on JCOS. The larger primitive data types float, double and long are unsupported while 32-bit integer support is only optional. Other unsupported features include multi-dimensional arrays, strings, threading and cloning. Performance-wise, JCOS offers slower execution speed when compared to SC operating systems that execute machine code directly. This is a inevitable characteristic of interpreted languages when compared to compiled ones like C. On many SC implementations this does not impact the user experience, as the performed routines generally do not take a perceivable amount of time. This does however change when asymmetric cryptographic algorithms need to be performed, especially with longer keys. Cryptographic coprocessors can counteract this problem but only work with certain algorithms. Hashing algorithms like the various versions of SHA are only implemented in software on current SmartCards.

2.9.6 Benefits of SmartCard and JavaCard based XMSS

Implementing XMSS on a SmartCard or JavaCard comes with the problem of slow execution times and tight memory restrictions. However, using these technologies also offers a number of benefits compared to alternative hardware.

Security

SmartCards offer a high level of security against both hardware and software based attacks. Thunstall [Tun17] lists three categories of attacks:

- Invasive Attacks:
  Reuire the SC chip to be removed from the card
- Semi-Invasive Attacks:
  Require an exposed surface of the chip
- Non-invasive Attacks:
  No physical damage to the SC

On a Hardware level, SCs are protected by physical barriers, scrambled chip design and hardware-based encryption. Even if an invasive or semi-invasive attack is performed, extracting useful information still is time consuming and costly. Non-Invasive methods like side channel attacks in the form of timing, power or electromagnetic analysis are possible in theory, but are also challenging to perform, even more so if countermeasures are taken on a software level. SmartCards and JavaCards are also closed systems that do not allow the injection of malicious software, therefore eliminating this security risk that classical computers are vulnerable to. Some institutions also require certain secrets so be stored on dedicated hardware modules like SmartCards. By executing operations that require the sk solely on the SC, it is never exposed to the outside world, maintaining a high level of security.

Cost

Compared Hardware Security Modules (HSM), that offer a similar level of security, the cost for a SmartCard based safety anchor is considerably lower. While a powerful HSM offers far better performance, they come at a cost of multiple thousand dollars. In cases where this performance advantage is not strictly necessary, SC can be considered as a more economical alternative.

2.10 Related Work

A number of previous works have researched the problem of generating hash-based signatures on devices with limited resources. A common issue with the principle of these schemes is the high computational cost compared to classical signature schemes like RSA or ECDSA.

2.10.1 XMSS with off-card key generation

Rohde et al. [RED$^+$08] were one of the first to implement MSS on a SmartCard. They used an 8-bit AVR processor, capable of running at 16MHz with an AES capable coprocessor. For their implementation they limited their tree height to $h = 16$ and therefore 2^{16} signatures in order to not exceed the maximum number of guaranteed erase/write cycles of the card's EEPROM, as the updated sk is written to this memory with every generated signature. For tree traversal the BDS algorithm described in 2.8 was used. In order to speed up the execution of hash functions they optimized an existing AES implementation that is capable of running on the coprocessor. They used a Matyas-Meyer-Oseas construction in order to generate a one-way encryption function. In essence this works similar to a hash function, as this function accepts an input of arbitrary length and outputs a message digest of a fixed size. This AES-based one-way function was used in most cases with an output length of 128bit. For cases where a collision resistant hash function is required, they also implemented a 256 bit hash function using AES. They argue that dedicated hash functions like SHA256 or MD5 are not well suited for MSS, as they operate with large block sizes which are not optimized for the short inputs and outputs mostly used in MSS. While SHA256 requires around 54K cycles to generate a 256 bit output from an input of equal size, their functions only require around 8K and 16K respectively. Using hardware acceleration by running these AES based functions on the coprocessor, the required clock cycles are further reduced to under 1K cycles for the main processor. As they argued that on-card key generation was not computationally feasible, they used a standard PC for key generation. For the hardware accelerated signature generation they measured times between 300 and 500ms, depending on the parameters chosen, exceeding the time required for RSA.

2.10.2 Security risks of AES-based hash functions

While these AES based hash functions do perform better than conventional hash functions, their usage entails security risks that prohibit their usage. Like Biryukov et al. [BK09] showed, related key attacks on AES are possible. When used as an encryption algorithm, this has no significant impact on the security of AES. When used as a hash function however, especially in the context of a hash based signature scheme where a lot of operations are performed and their value is exposed publicly, the security of AES based hash functions are not sufficient.

2.10.3 Forward Secure Signatures on Smart Cards

Huelsing et al.[HBB13] introduced an extension to XMSS called XMSS+. This scheme reduces the complexity of key generation from $\mathcal{O}(n)$ to $\mathcal{O}(\sqrt{n})$. XMSS+ builds upon $CMSS$ [BGD+06], a predecessor to XMSS$^{\mathrm{MT}}$ that basically works similar to XMSS$^{\mathrm{MT}}$ with $d = 2$. The bottom layer of trees are used to sign the actual messages while the top layer is used to sigh the root nodes of the bottom layer. They fully implemented this forward secure scheme on a SmartCard, including the key pair generation that Rohde et al. sourced out to a PC. As the BDS algorithm does not always require the full $(h - k)/2$ updates to perform all updates required in the respective cycle, the remaining updates are used to precalculate the next tree required on the bottom layer. They used a SmartCard with a 16-bit CPU running at 33MHz. Similar to Rhode et al. they also implemented the function families F and H based on AES in order to use the card's cryptographic coprocessor. They tested different tree sizes of $h = 16$ and $h = 20$ with different parameter sets for w and k. Building the initial tree, including pk and the first authentication path took between 5,6 and 10,5 seconds for a tree of height $h = 16$ depending on the parameters used with singing times below 200ms in all those cases. They showed that signing times of XMSS+ are similar to XMSS but key generation is considerably faster, allowing an implementation fully on the SmartCard.

This work followed Christoph Busold's Master Thesis [Bus12]. He implemented a forward secure version of XMSS and XMSS+ on SmartCards and put a focus on the memory management of this scheme. As the number of write-cycles to non-volatile memory are limited, these need to be minimized. He analyzed write cycles for both schemes with the result being that the number of cycles performed in his implementation does not exceed the card's limit. In addition the amount of memory required is small enough to allow moving it to different memory cells if needed.

2.10.4 XMSS on JavaCards

While the previously presented works used SmartCards programmed either in C or Assembler, van der Laan et al. [LPR+18] implemented XMSS$^{\mathrm{MT}}$ on JavaCard based SCs in order to study whether this technology is capable of running hash based signatures. Instead of using BDS tree traversal they stored all nodes of the currently active tree in memory together with the next one required on the respective layer. Whenever one tree is consumed, the cached one is used and a new tree is constructed. Instead of performing all the steps necessary for a XMSS signature generation in a single computation they split it up into steps, each triggered by a command APDU. Figure 2.7 shows these states. After completing the computation of a phase, the card sends a response APDU holding parts of the signature. After the output of the last part of the signature, the preparation phase is triggered, where a new WOTS+ pk and the next authentication path is constructed. This approach has the benefit of allowing the card to perform more operations on RAM only. Most cards do not have enough RAM to store the WOTS+ signature yet alone the whole XMSS signature. So either the slow EEPROM must be used to store this data or partial calculations are performed and the signature is only fully assembled on the application using the card. Their implementation therefore follows the latter approach. For their performance measurements they use the parameter set $n = 32$ bytes and $w = 16$. For a single WOTS+ signing operation this results in an average runtime of 33 seconds. For a tree of height $h = 20$ with $d = 4$ leaf nodes and therefore 32 signatures per subtree a complete signing operation with the authentication path already pre-computed

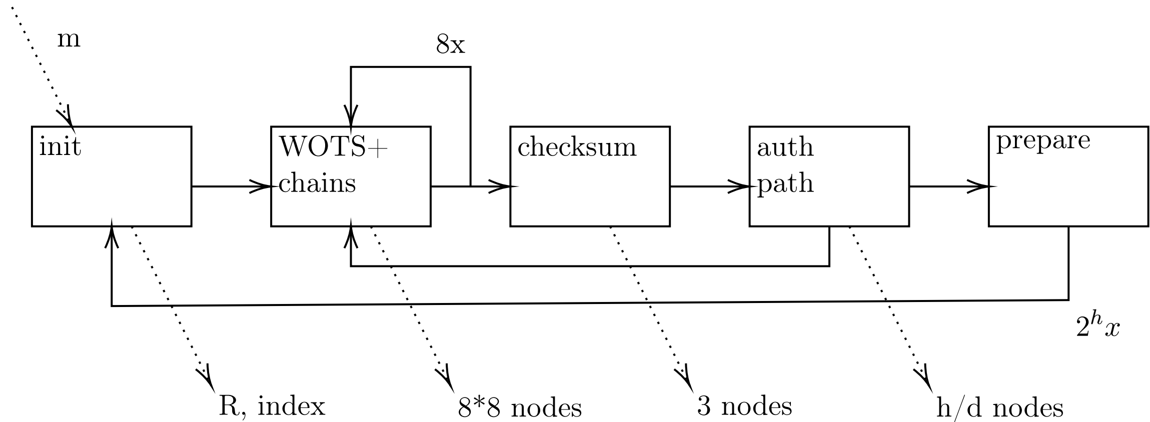


Figure 2.7: State diagram as shown in [LPR+18]. Dottet arrows indicate APDU communication. Command APDUs triggering the single state computations are not shown.

takes ≈ 54 seconds in the best case. The preparation step takes 85 seconds in the best case. As this implementation computes a new leaf whenever one is consumed, an additional WOTS+ computation is performed every 32nd signature when a leaf is consumed on the second tree layer. Every 256 signatures another additional WOTS+ signature is required on the next tree level and so on, thus further increasing the runtime. They also devote a large section of the paper to potential improvements to XMSS. They propose potential ways to increase the speed of the hash function calls through parallelization. For example instead of performing $len * (w - 1)$ individual hash function calls, only $(w - 1)$ calls to a function accepting len blocks of input data could be performed, thus decreasing the cost of passing through the Java stack. They also discuss different levels of abstraction the applet could provide, ranging from their approach of triggering individual subroutines up to a high level of creating whole XMSS^{MT} signatures by a single command APDU.

2.10.5 Post-quantum security on other constrained devices

QuantumRISC [4] is an active research project dedicated specifically to bringing post quantum security systems to constrained devices. A very recent paper by Campos et al. [CKRS20] compared implementations of XMSS and the Leighton-Micali Hash-Based Signature Scheme (LMS) [MCF19]. LMS is similar to XMSS, with both using a Merkle tree for the key management and a layered tree structure. They modified the reference implementations of both schemes in order to speed up the implementation times. Their proposed improvements include replacing the L-tree compression with a single hash call and discarding the use of bitmasks in the OTS chaining functions. The resulting scheme they call XMSS_SIMPLE is similar to LMS, which also follows these design choices. Their implementation achieved a speedup of factor 3.11 in comparison to the reference implementation. LMS is only proven in the Random Oracle Model [Eat17], while XMSS is proven to be secure in the standard model [BDH11] under minimum security assumptions [PK17]. The proof in the standard model depends on the L-tree structure. As a result the scheme implemented by Campos et al. does have stronger security assumptions than XMSS, which is undesirable. As this paper

[4] https://quantumrisc.de/index.html, visited 27th of July

was published shortly before the date of submission of this work, a comparable approach could not be implemented.

3 Methodology and Concept

In the following chapter, the methodology of this work will be described. This includes the hardware choice, consideration of requirements for the proof of concept implementation and the deduction of possible use-cases.

3.1 Requirement Analysis

As already mentioned in Chapter 1, the advancement made in the field of quantum computing require new cryptographic functions. Just like in conventional security systems, different use cases require different levels of security and just like in some cases secure hardware modules like SmartCards are required, the same is true for post quantum cryptography. In order to better understand the requirements a SmartCard based quantum secure signature algorithm has to fulfill, some use cases that already use card based signing operations where considered. In essence, every application that requires a SmartCard for authentication or any application that requires a separate secure hardware token to store keys can be considered as a possible use-case. Following is a list of cases further examined:

- E-Mail and Data exchange
- VPN Authentication
- VPN Gateway Authentication
- User Authentication
- Online Banking

Each application requires slightly different properties. In some cases the size of signatures is important while others require the fastest possible runtimes. Also the number of signatures that can be generated by the scheme has to be sufficiently large. Conventional digital signature schemes like RSA allow the user to have a single key pair and use it so generate a practically infinite number of signatures. The same cannot be said about XMSS, as it has a limited number of signatures by design. As a result, a sufficiently large tree must be supported by the hardware in order to circumvent this restriction. As to what can be considered a sufficiently large number depends on the use-case but in general the tree should not run out of signatures during the lifetime of the scheme, which again is different for each use-case. In order to have a reference, lifetimes as they are usually defined for key pairs of conventional signature algorithms are considered. Another important feature is the construction and storage of all keys on card. This is a requirement in some use-cases and is therefore considered a mandatory feature [EPC20, GCHW15]. In general the goal is to minimize any negative impact introduced by the usage of XMSS instead of conventional signature schemes.

3.2 Requirements for the implementation

For an implementation to fulfill the above mentioned requirements, a number of things have to be considered. As already mentioned in Section 2.10, previous approaches already encountered the problem of expensive on-card key generation due to limited processing powers available on SmartCards. However, on-card key generation without exposing secret keys to hardware outside the card is required in order to achieve a security level similar to conventional schemes also running solely on SmartCards. As a result, focus should be laid on optimizing the implementation for minimal runtimes. Another factor to consider is the fact, that not all SmartCards offer the same memory capabilities. In order for the application to be applicable in the real world, memory requirements should be minimal in order to be executable on a wide range of card models. Any memory space left on the card could then be used for node storage in the way described in Section 2.8. In general, the implementation should be flexible enough to allow easy changes to the configuration. This is not only important for testing purposes but also for deployment in a practical environment, as this allows potential optimizations depending on the use-case.

3.3 Advantages of JavaCard

A non-functional but also important feature the implementation should provide is the ability to be easily deployable. For an application to be practicable in the real world, it should not come with major restrictions as far as hardware requirements are concerned. This is an advantage of the JavaCard system over other SmartCard operating systems as compatibility of applets is manufacturer independent. As applets are executed inside the JavaCard VM, almost every card supporting the required version of JCOS is able to run the same CAP file. The exception to this rule are low-end cards with insufficient memory. This is where the ability of easily configuring the application mentioned in Section 3.2 becomes important. With lower level programming languages, bigger modifications are necessary for each SC processor type, thus hindering a potential widespread distribution. Another reason for JavaCard is the wide range of API functions, especially for cryptographic functions. These allow the programmer to work on a higher abstraction layer compared to C or Assembler implementation, therefore accelerating the development process and generally producing more readable code.

3.4 The Hardware choice

In this work a NXP JCOP3 J3H145 JavaCard with a SmartMX2 P60 Controller [MoT19] is used for testing. This card runs at a maximum of 5MHz and offers 132 KByte of user-accessible EEPROM and 3,7 KByte of RAM. Its CPU is comparable to other popular cards, therefore allowing conclusions to the expected real-world performance of the implementation. With the EEPROM being of a comparably large size for a typical JavaCard, this also allows the examination of larger XMSS trees than necessary for most use-cases.

4 Implementation

In order to test an implementation of $XMSS^{MT}$ on current JavaCard technology, a version close to the reference implementation [1] was implemented. The developed applet is compiled for JCOS 3.0.4 and requires 31KByte of storage. All hash functions were implemented using SHA-256, as it offers adequate performance for a cryptographic hash function and returns results of 32 byte length, therefore providing a suitable security level. This way, a comparison to related works where 32 is also a popular choice for the parameter n is also possible. The usage of a hardware accelerated hash function as used in other SC based XMSS implementations is not possible for the reasons described in Subsection 2.10.2. In order to be able to provide an adequate amount of signatures, $XMSS^{MT}$ is used. As tree traversal the BDS algorithm described in Section 2.8 was chosen due to the advantages mentioned there. Other versions not utilizing any tree traversal optimization and a single tree variant were also implemented for testing purposes.

4.1 Development steps

Most of the development was performed inside a simulator environment. To this end, the official Eclipse plugin provided by Oracle was used. A first implementation only used a naive way to construct authentication paths, calculating every node once it is required. As described in Section 2.7, this results in a re-evaluation of almost all XMSS nodes, only omitting the current leaf node and the hash tree root. This can be considered a best-case memory and worst-case runtime implementation, as no memory is required for storing any tree nodes with the cost of having to reconstruct almost the entire tree for each signature. First tests already showed that this implementation is not feasible for real-world usage, as already a tree of height $h = 4$, providing 16 possible signatures, required more than a minute for a single signing operation. In this case, the runtime is approximately doubled for every height increment. With every increment the number of leaves doubles and therefore the number of WOTS+ key pairs necessary for the tree construction. The number of nodes that are constructed also doubles with one additional operation required to construct the root node on the new top layer. This rough estimation was shown to hold true for different tree sizes tested and will be further discussed in Subsection 5.3.3.

In a second phase, the BDS tree traversal algorithm was implemented for the single tree variant of XMSS. The time/space trade-off provided by BDS through its parameters can be analyzed this way to find a suitable parameter set for each card depending on the memory available. Using the parameter k, this implementation also allows to analyze a close to worst case memory, best case runtime strategy by storing all right tree nodes in memory during the initial pk construction. While this algorithm already allows faster signature generation, the time required for the initial signature generation is slightly worse, as nodes need to be stored in permanent memory while the number of leaf and node evaluations remain equal.

[1] https://github.com/XMSS/xmss-reference, repository as of 12th of April was used as a reference

This operation only needs to be performed a single time but this is still a constraint with larger trees. For a tree of height $h = 12$, which is able to generate 4096 signatures, this initialization already takes approximately 24,5 hours. In order to reduce this initial runtime while also increasing the number of available signatures, XMSS^{MT} was implemented. This multi-tree variant also uses the BDS algorithm with separate instances for each tree. This way, trees of height $h = 20$ and higher can be used, whose pk construction would not be computationally feasible on a JavaCard with only a single tree.

4.2 JavaCard programming guidelines and best practises

In general there are a number of pitfalls that need to be avoided and programmers are likely not used to when coming from conventional programming in C/C++ or Java. Those became apparent when test where performed during the above described development steps. Before the final implementation is described, this section introduces a number of pitfalls to avoid and best practices to follow. This allows a better understanding of some design choices followed in the final implementation. A number of best practices where adopted from a document provided by ruimtools [2].

Reusing objects

As already described in Subsection 2.9.4, objects allocated on the heap have a lifetime that begins with the initialization of the card end ends with its destruction. Consequently, only primitive data types can be allocated inside functions, all other objects, including arrays, must only be created exactly once. Due to memory limitations, simply creating objects solely for usage within specific functions is not possible and reusing arrays between different functions is necessary. Consequently, the programmer has to keep track which data blocks are used within functions in order to prevent overwriting data that is required later. Another important factor when it comes efficient usage of the memory is the fact that EEPROM is managed in 32-byte chunks. Each one of them can only be used by a single object. So for example if a byte array of 8 bytes would be allocated, the actual memory requirements of this object amount to 32 bytes. Furthermore, the JCOS VM requires 8 bytes of additional storage for each primitive type array. So where normally a large number of small arrays would be used, the allocation of a single memory block and working with offsets is advisable.

Call stack exhaustion

Another thing to avoid is the repeated usage of small functions, as this introduces overhead through exhaustion of the Java call stack. For example the keyed hash function F is called multiple times from within the chaining function. While implementing F as an individual method is convenient, this should be avoided and the routine should instead be implemented directly inside the chaining function. Following this practice does mean that some functions will be implemented at multiple places inside the code and the size of the executable might increase slightly, but doing so reduces the overall runtime. Whether this strategy should be applied depends on the individual case. Testing showed that in some places the overhead introduced by executing the subroutine as an individual function was negligible. In other

[2] http://www.ruimtools.com/doc.php?doc=jc_best, website visited 19th of July 2020

cases, like inside the chaining function, speedups justifying a direct implementation could be achieved.

Flat class hierarchies

An overhead similar to repeated function calls is introduced by the usage of deep class hierarchies. With every level of inheritance the cost of method invocation increases. As a result the deepness of the class hierarchy should be minimized.

JCOS API

An advantage of the JCOS is the wide range of native APIs. These should be used whenever possible. This is particularly important when operating on arrays. Here the *Util* framework [3] provides a number of functions that allow fast operations on arrays. The implementation should also consider a design that allows the usage of the correct version of the functions provided. For example the hash functions provided by the MessageDigest class [4] can be invocated in different ways. If the input data is distributed over multiple arrays, the *update* method has to be called for each one of them, thus requiring additional computation time and memory. A better way to hash input data is to design the program in a way than input data is already accumulated in a single array at the time it is hashed. This way a single call of the *doFinal* method is possible, avoiding copy operations and additional memory consumption.

4.3 Description of the final implementation

The final implementation used for testing purposes was programmed generally following the reference implementation. However, a direct conversion of the code written in C to JavaCard is not possible due to constraints of the JCOS. In order to optimize the implementation for running efficiently on JavaCards, optimizations described in Section 4.2 are applied. The only class instantiated is the global applet class, thus minimizing the overhead for any method invocation. Where the reference implementation uses structs, for example to hold the different BDS instances, the JavaCard implementation uses a single plain byte array in order to minimize the overall memory consumption as described in Section 4.2. In order to reduce the overall runtime, EEPROM accesses need to be prevented wherever possible. However, working only with the available RAM was not possible, as the storage requirements exceed the limits of the available memory. This has two reasons. First, some structures, for example the array holding the final signature exceeds the available RAM alone. Secondly, the BDS and treehash states need to be stored in persistent memory, thus requiring writing operations on EEPROM. In order to be able to compare the effect that different parameter sets have on signing times and sizes, the implementation can easily be modified. A number of globally defined variables allow the configuration of the tree dimensions as well as the BDS parameter sets.

Provided Signing API			
Name	**ID**	**Response**	**Description**
INIT_PK	0xAA	public key	Initialization
SIGN_EXT	0xA0	XMSS sig.	Signing + auth. path preparation
SIGN	0xA1	size of sig. in bytes	Signing + auth. path preparation
GET_SIG	0xA2	255 bytes sig. fragment	used for signature reteival
SIGN_ONLY_EXT	0xA3	XMSS sig.	Signing operation only
SIGN_ONLY	0xA4	size of sig. in bytes	Signing operation only
AUTH_ONLY	0xA5		auth. path preparation only

Table 4.1: Signing API provided by the card. None of the functions expect any value in the fields P1 and P2. The APDUs triggering the signing operations require the 32 byte message in their data field. AUTH_ONLY does not expect a response.

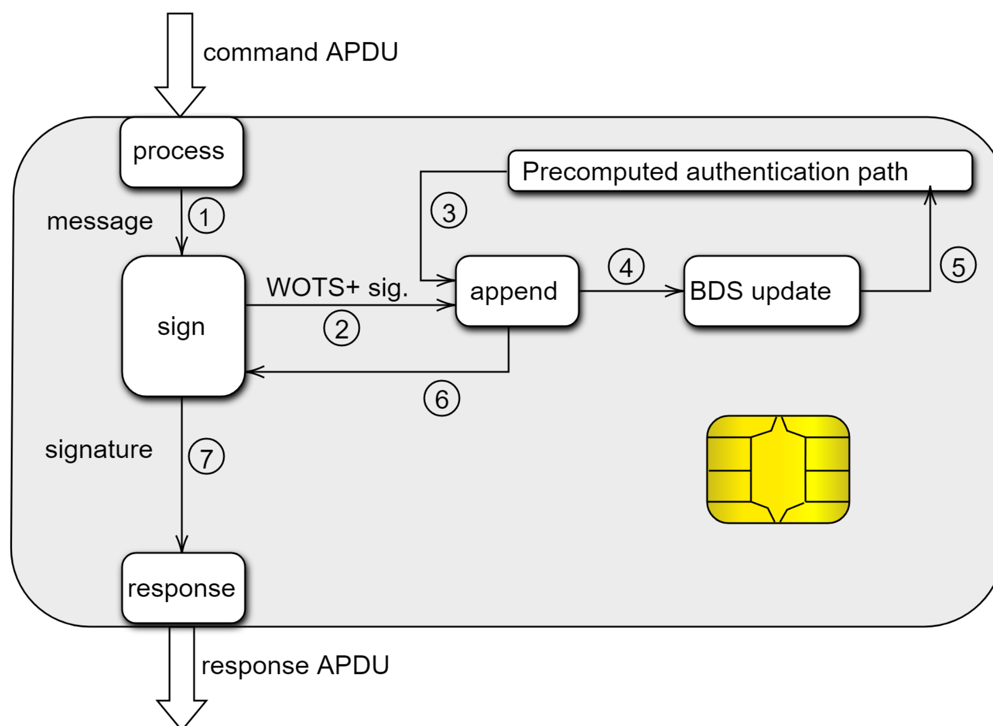


Figure 4.1: Visualization of a signing operation with authentication path preparation. The received message is handed over to the signing function ①. Here the WOTS+ signature is created ② and the authentication path computed in the precious round is appended to it ③. Before the now complete XMSS signature is returned, the authentication path for the next round is computed ④ and stored im memory for the next round ⑤. Then the signing funktion returns the XMSS signature ⑥. It is then forwarded to a response function ⑦, which either returns an APDU containing the signature or sends its size if extended APDUs are not supported.

4.3.1 Provided API

User interaction through APDU commands was designed to provide high level functionality and hiding most complexity from the user. The command APDUs required for using the card are listed in Table 4.1. Before a signature can be created, the public key must be constructed. The INIT_PK APDU is therefore sent to the card. The *pk* is returned by the card as a response after a successful initialization. In order to create a signature, only a single APDU holding the data is sent to the card. The general flow of a signing operation is visualized in Figure 4.1. The signing operation consists of two major parts, the first one being the construction of the current signature and the second one being the preparation of the authentication path for the next round. This design allows a separation of these two steps. SIGN_ONLY or SIGN_ONLY_EXT command APDUs can be used for faster signature generation. In this case steps ④ and ⑤ of the routine shown in Figure 4.1 are not executed and the signature is returned directly after its construction. Before another signature can be created, the AUTH_ONLY APDU must be sent to the card, thus triggering the preparation of the next authentication path by executing the steps skipped previously. From now on this approach is referred to as the two-step approach with the usage of the SIGN or SIGN_EXT APDUs being referred to as single-step signing. To increase compatibility with different SC readers, there are two signing modes. If extended APDUs are supported, SIGN_EXT can be used to initiate a signing operation. The signer then receives an APDU containg the complete XMSS signature. A second mode for readers not capable of sending and receiving more than 255 bytes of payload is available. In this case, the response APDU holds a number, indicating the signature size in bytes. It is then up to the application to retrieve the signature in chunks of 255 bytes by sending the GET_SIG APDU for the appropriate number of times, witch each response APDU containing the next 255 byte fragment. If the signature size is not a multiple of 255, the last chunk holds only the remaining bytes. The applet expects input messages of exactly 32 bytes. In most cases it is therefore necessary to apply a hash function with 256bit output to data that needs to be signed. While it would be possible to transfer data directly to the card, this creates problems in practice. First the transfer of a larger amount of data requires more time. Secondly, any input to the applet needs to be hashed on-card. This operation also requires more time with larger input data, especially in contrast to the same operation executed beforehand on a regular PC.

In addition to the API specific to XMSS, a number of other functions were implemented. In order to measure the performance of JC API functions, including copy operations and hash functions, a method for running those in a loop for a selected number of times was implemented. For the runtime analysis of the subroutines performed during a signing operation, these can also be executed directly.

4.4 Limitations

The final implementation has a number of shortcoming. Through the requirement of storing portions of the BDS and treehash states on EEPROM, writing operations during every signing operation are performed on it. This is problematic as this memory technology only allows a limited number of total write operations. A minimum number of 10^4 write operations

[3]docs.oracle.com/javacard/3.0.5/api/javacard/framework/Util.html, website visited on 19th of July

[4]https://docs.oracle.com/javacard/3.0.5/api/javacard/security/MessageDigest.html, site visited 19th of July

per cell is an usual specification [BZ08]. Li et al. [LYH11] confirmed that EEPROM cells start to degrade at his point. The current implementation does not consider this. Possible approaches to circumvent this problem would be to realign often usd variables inside the array structures, thus changing their memory addresses. An easier approach could also be the allocation of additional arrays. After a certain amount of signatures the contents could be copied over to the fresh array, while the old memory block is no longer used.

While the implementation is able to construct large trees and create correct signatures for them, the actual amount of signatures that can be created is limited to 2^{15}. This is due to the usage of a short value to keep track of the current signature index. Previous implementations used a pair of short values, one storing the lower 15 bits and one storing the higher 15 bits, thus being able to represent numbers of up to 2^{30} [LPR+18]. This approach performs a conversion from the two shorts into a byte array two times per signing operation. Implementing the same functionality for the scheme presented here would in theory be possible but was not further considered as the thereby introduced complexity was not considered worthwhile. The introduced computational overhead should not have a significant impact on the overall performance. In places where the index is used repeatedly, like for example as part of addresses used as input for the hash functions, it is beforehand converted into a 4 byte representation anyway. Usages outside of address bytes are limited to indexing subtree leaves. Here the limitation of the short data type can be neglected and a single short value could be used, as only subtrees of height $(h/d) = 15$ would cause an overflow. However, using a subtree of this size is not sufficient anyway for reasons described later in Subsection 5.3.3.

The implementation does also not offer any form of countermeasure against card tears. In a real-world environment it must be assumed that users could remove the card during a signing operation. In case such a card tear happens, the calculations of all BDS and treehash instances must be finished before the next signature can be issued. It would therefore be necessary to implement atomic operations in some places in order to make sure that updates to these instances are performed correctly. As far as performance is concerned, it is to be expected that implementing these would result in longer runtimes.

4.5 Test and Verification

The communication with the card was performed using the Java Smart Card I/O API. The runtime measurements where performed using Java test cases with a timing function starting immediately before sending the command APDU and ending with the reception of the response. Signatures where created using the single-step approach. The built-in System.currentTimeMillis() function was used to obtain the respective timestamps, providing millisecond resolution. Although the resolution of this function depends on the operating system and can therefore be of a lower granularity, this should not have an negative impact on the quality of the measurements, as even the lowest times are of lengths of several seconds, relativizing inaccuracies.

4.5.1 Variations in WOTS+ signing times

The WOTS+ signature calculation could take approximately the same amount of time as a WOTS+ pk generation. As described in Section 2.6.1, the input message is mapped to values between 0 and $(w-1)$. If the hash value of the input message, generated by H_{msg}, would

now consist entirely of values that get mapped to $w-1$, the chaining function would in turn also be executed $w-1$ times on each segment. The same could also happen in a best case scenario, where every segment gets mapped to 0, thus requiring no chaining function calls. This fact introduces variations in signature times depending on the input message. While the output of SHA-256, the function used for the construction of H_{msg}, cannot be expected to output values with a perfectly random distribution [AAK19], measurements showed that the average runtime of a signing operation was ≈ 9.79 seconds. In order to be able to analyze the signature runtimes solely based on the parameter sets used, this randomness was factored out for tests. To this end, the same message was used to generate all signatures used for the evaluations. Random values used for the creation of seeds showed to not have an measurable impact on runtimes, as they are only used as pseudorandom function input.

4.5.2 Verification

In order to test the correctness of the JavaCard implementation, the signatures produced by the card were verified against the XMSS reference implementation [5]. To this end, a test program's output was written into a log file. The first line of this file holds the *pk* and each following line a signature. The reference implementation was modified to read it and verify the signatures against the provided *pk*. A verification of the correctness was performed for all parameter sets used for the evaluation, both with the message used for the evaluation as well as with randomly generated ones. For these random tests, 32 byte arrays where generated using an online random number generator[6]. Each byte array was then packed into a command APDU, signed multiple times and verified against the reference implementation.

[5] https://github.com/XMSS/xmss-reference, Version at April 12th 2020, https://github.com/XMSS/xmss-reference/tree/fb7e3f8edce8d412a707f522d597ab3546863202

[6] https://www.random.org/bytes/

5 Evaluation

In the following section, the results of the timing measurements will be presented. In the beginning, the general SC performance for single operations will be shown. After that, the actual XMSS signature generation times will be discussed. The general communication overhead, meaning the time required for sending an APDU with 32 byte payload and receiving a response was measured to be less than 10ms.

5.1 General JavaCard performance

In this section, the general performance of the used JavaCard will be presented. As shown by Erdmann [Erd04], performance varies between manufacturers and models. The times presented later in this chapter are therefore, at least to a certain extent, only significant to the card used in testing. The results for other models should not deviate far from the measured results, especially as the implementation does not take advantage of specific hardware modules like cryptographic coprocessors. Nonetheless, this still is a factor to consider. This is intended to offer a performance baseline for comparison with future work and independent from the actual XMSS implementation. The chosen functions and their input and output parameters are the ones most relevant for XMSS. The hash function, in this case SHA-256, has the biggest impact on the XMSS performance. The array copy functions are also often used, as they are another building block of the keyed hash functions. The most relevant input lengths for the hash functions are $2 * N$ and $3 * N$ bytes, in this case 64 and 96 bytes, as those are the ones used in the functions F and H. Everything beyond that is only used once in XMSS, during the initial hashing of the XMSS input data in the function H_{msg}, which should be avoided for a SC based implementation anyway for the reasons mentioned in 4.3.1. An important factor is the type of memory those functions operate on, as RAM is significantly faster.

The results for SHA-256 are listed in table 5.1. The functions where called 1000 times with the respective input lengths in bytes. The presented results are the average values of 10 runs. Input and output arrays do not overlap, although they may do so as testing showed that this has no performance impact. The hash function was always directly executed using the

SHA-256		**bytes**			
from	**to**	**32**	**64**	**96**	**128**
RAM	**RAM**	2503	4023	4117	5668
EEPROM	**EEPROM**	11225	12779	12878	14435
RAM	**EEPROM**	2885	4438	4523	6810
EEPROM	**RAM**	2494	4470	4149	5707

Table 5.1: Runtimes for 1000 iterations of SHA-256 on different input lengths, runtimes in milliseconds

from / to	RAM	EEPROM
RAM	3280	8691
EEPROM	3354	9204

Table 5.2: copy 128bytes, 10.000 iterations, runtimes in milliseconds

$MessageDigest.doFinal()$ method, as performing $MessageDigest.update()$ to collect input data can have a negative performance input and should therefore be avoided if possible[1]. When comparing the performance of SHA-256 when executed solely on RAM and EEPROM respectively, RAM is faster by a factor between 2,54 for 32 byte input and 4,5 for 128 byte input. When hashing data from EEPROM to RAM, runtimes are equal to the ones solely on RAM to within a margin of error. The other way round with hash outputs written to EEPROM, the cost is significantly higher. For copy operations, 128 bytes where copies 10.000 times from array to array. The results listed in table 5.2 show that writing to EEPROM has the highest impact on performance. Reading from EEPROM and writing to RAM is only slightly slower than operating on RAM directly.

5.2 Splitting the signing and preparation steps

As described in section 4.3.1, the API also allows splitting the signature generation and authentication path preparation. This way signing operations can be performed faster, as the preparation for the next signature is skipped. This however requires the host application to schule the preparation step afterwards. This step does not require any input message and can therefore be executed completely in background. Using this two-step signing approach allows faster signature generation if only a single signature is required at a time. If however a series of signatures is requested from the card, no speedup in comparison to the single-step operation can be achieved, as the next signature can only be computed after the preparation step is finished. The signature generation alone requires $\approx$ 10.87 seconds on average, depending on the input message as described in Subsection 4.5.1. The time required for the construction of the authentication path depends on the parameters used. The total on-card runtime of the two-step approach was measured to be $<$50ms slower compared to executing them in a single computation.

5.3 Finding appropriate parameter sets

As mentioned in Section 2.8, the parameters of the BDS algorithm offer a trade-off between memory consumption and speedup of signature generation. Additionally, the Winternitz parameter w offers a trade-off between signature size and time. The following section describes the impact of these parameters and the way they where chosen.

5.3.1 Winternitz parameter *w*

The Winternitz parameter w theoretically can be chosen freely but is recommended to be of size 4 or 16, as these values offer a good trade-off between runtime and signature size. RFC-8391 [HBG+18] defines different parameter sets for XMSS with $w = 16$ and combinations

[1] https://docs.oracle.com/javacard/3.0.5/api/javacard/security/MessageDigest.html

Tree Structure	**h=6 d=2**		**h=20 d=5**	
w	**4**	**16**	**4**	**16**
init time	351491	501520	1744764	2464181
average	41791	59745	44032	62256
signature size	8737	4513	21.955	10775

Table 5.3: Impact of w on signature times and sizes; times in milliseconds, sizes in bytes

	$w = 4$	$w = 16$
WOTS sign	9792	14468
WOTS pk gen.	12289	25356
L-tree construction	6174	3210
XMSS leaf gen.	19186	28874

Table 5.4: Impact of w on subroutines, runtimes in milliseconds

of values for n, h and hash functions. Parameter sets for values $w \neq 16$ are not officially defined and support must therefore be implemented explicitly. While XMSS implementations running on more powerful devices may prefer smaller signatures over a speedup, the biggest restraining factor for a JavaCard based version is the long runtime. The results presented in table 5.3 show the impact of the parameter w for different tree sizes.

By using $w = 4$, the cost of WOTS+ key generation and the signing operation is reduced. The impact can be better seen when comparing the runtimes of the individual subroutines. Table 5.4 shows runtimes for the relevant functions. Especially the time required to compute a WOTS+ pk using the chaining function is reduced significantly. Consequently, also the time required for the WOTS+ signing operation is reduced. The hash tree itself is not affected by this parameter, as the L-trees compress the WOTS+ keys to n bytes regardless of their length. Due to the increased length of the WOTS+ pk, this compression takes longer for $w = 4$. However, this is more than compensated for by the faster chaining function runtime as can be seen in the XMSS leaf generation time in the last row. In conclusion, for purely JavaCard based implementation of XMSS, $w = 4$ is the better choice as signature sizes are less of a critical factor than signature generation times. There might be cases where the signature time can be hidden from a user, but in most cases a faster runtime if favorable.

5.3.2 Impact of parameter k

The BDS parameter k is used to control the number of top layers, where right child nodes are stored during signature generation, providing a speedup with the cost of additional memory consumption. This parameter can be set accordingly to the specifications of the card. For XMSS$^{\mathrm{MT}}$, the memory requirements for this parameters are moderate, as even storing all right nodes of all currently active trees only requires $d * (2^{(h/d)} - 2) * n$ bytes of memory. For a multi-tree of height 20, with $d = 5$ subtrees of $(h/d) = 4$ height, this results in a memory cost of 2240 bytes in EEPROM. The actual memory consumption varies between different tree configurations, but if possible the parameter k should be set as $k = h$. To illustrate this, Figure 5.1 shows the runtimes of the first 76 signatures of an XMSS$^{\mathrm{MT}}$ tree of height 20 with subtrees of height 4. For the value of $k = 4$ the average, best- and worst-case runtimes exceed those of lower values for k. The spikes in signatures 16 and 48 of the $k = 0$

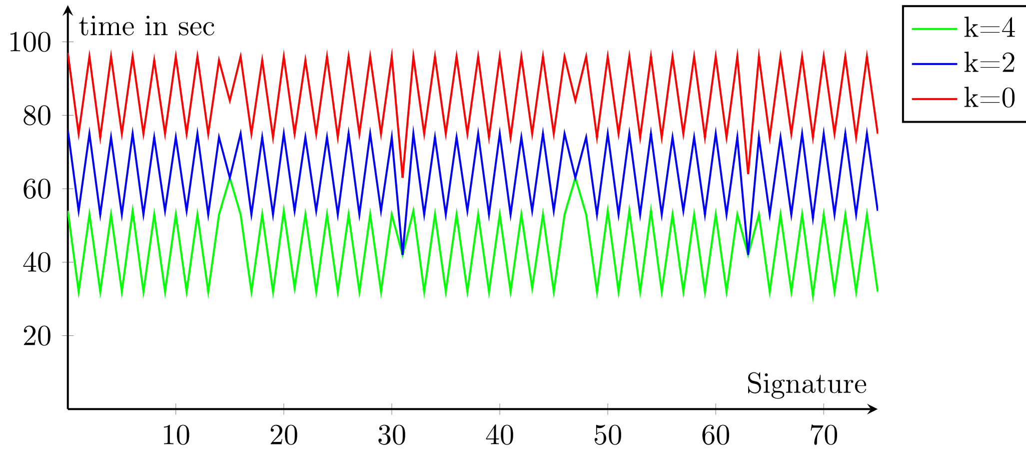


Figure 5.1: Signing times for different parameters k in the first 70 signatures of an XMSS^{MT} tree of height $h = 20$ with $d = 5$ subtrees

Tree Structure	h=6 d=2		h=20 d=5	
k	0	4	0	4
average	47.68	41.79	85.24	43.69
best-case	10.04	10.02	63.09	31.12
worst-case	74.88	64.02	96.80	63.26

Table 5.5: Impact of k on signature times, runtimes in seconds

graph are due to the switch of the active tree on the lowest layer. For the other graphs the cost of this switch is masked by the higher number of required WOTS+ pk per signing operation. As mentioned in 2.8.1, on average $(h - k + 1)/2$ leaf nodes are computed with every signature. The fact that not every signing operation requires the same amount of leaf node computation becomes obvious in the graphs. For every second signature an additional leaf node is computed, resulting in those signatures requiring around 19 seconds longer on average, the same amount of time required for the construction of an XMSS leaf as presented in Table 5.4.

5.3.3 Subtree height and number

A factor when it comes to multi-tree XMSS is the ratio between the tree height and the number of subtrees. The number of subtrees d needs to be chosen so that is divides h without remainder. For trees of larger height h this leaves a number of options. For example for a tree of height $h = 20$ a valid option for d is any divisor of h and therefore any member of the set $D = 1, 2, 4, 5, 10, 20$. Setting $d = 1$ is basically XMSS without subtrees. A value of $d = 20$ is possible in theory, although it does not really make sense. In this case every tree node is considered to be a tree of height 1 with its own BDS instance, resulting in additional memory and runtime overhead in comparison to $d = 1$ without any benefits. The ratio between subtree height and the number of subtrees is another trade-off. If the same overall tree height is maintained, a higher value of d also increases the signature size as additional OTS signatures are included for every subtree. Decreasing d results in smaller signatures but

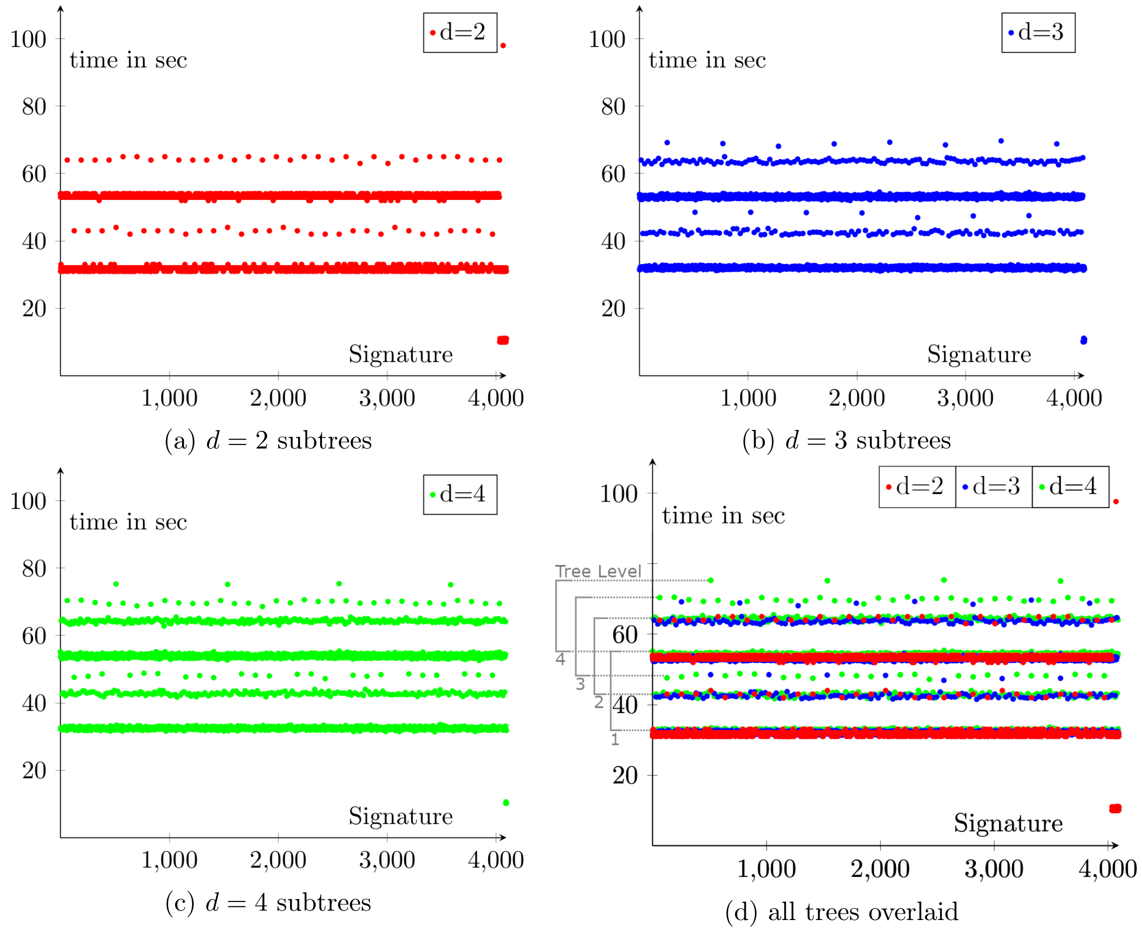


Figure 5.2: Comparison of three different ratios between subtree height and number. All trees are of height $h = 12$. Subfigure (d) with indicators for runtimes introduced by the respective subtree layer.

requires more memory for node storage, especially with larger values for k and also increases the time required for tree construction. Lower values of d also come with the benefit of less frequent tree switches due to the increased number of leaves of every subtree. For all the measurements in this section, the parameters $w = 4$ and $k = h/d$ are chosen, as those provided the fastest runtimes.

Full tree evaluation

In order to see the impact of d, a tree of size $h = 12$ was studied further. A valid value for d is in this case a member of the set $D = 1, 2, 3, 4, 6, 12$. The measurements were performed only using the subset $3, 4, 6$ as the other values are not viable with $d = 1$ requiring a non viable initialization time as mentioned in Section 4.1 and values beyond $d = 6$ requiring frequent tree switches due to a small subtree size. While a tree of height $h = 12$ offers only 4096 signatures and is therefore too small for most real-world use-cases, the impact of the ratio between d and the subtree height still becomes evident. In addition, the relatively small number of signatures allowed a complete run-through of this tree. The creation of all 4096 signatures took between ≈ 49 hours in the $d = 2$ case to ≈ 52 hours for the parameter $d = 4$. With the signature count doubling with every height increment, fully evaluating even larger trees was not considered worthwhile as the observation also becomes clear in in a tree of height $h = 12$. Figure 5.2d shows the signature times for the tested values of d for a tree of height $h = 12$. Clearly visible are the worst-case runtimes caused by a switch to a new right tree with $\approx 74sec$ and $\approx 70sec$ for $d = 4$ and $d = 3$ respectively. Another interesting observation is the fact, that the size of the tree that needs to be switched only has a minor impact on runtimes. The important factor for run times is the number of trees, as each one has its own BDS instance which in turn require WOTS+ pk computations as described in Subsection 2.8.2. These sum up for some signatures, thus requiring multiple pk operations for a single signature generation. The results of $d = 2$ again show an improvement to the average runtime as once again one subtree-layer less is handled. This again comes with a cost of more memory consumption and longer initialization time. An interesting note here is that signature 4056 was measured with a runtime of 98.12 second. Whether this was down to a problem within the PC/SC interface, a hardware problem on the card or a bug in the implementation could not be determined and was not encountered during any other measurement.

Subtree height conclusion

Whether the benefits of longer initialization times and higher memory requirements outweigh the cost depends on the use-case and hardware resources. Large subtrees quickly become too expensive. Table 5.6 shows estimated initialization times for different ratios between subtree height and the number of subtrees. Values not measured during evaluation where inter-/extrapolated. Further testing showed that the resulting estimations where well under 10% deviation. This also shows that initialization times of subtrees increase approximately linear in their height. The same is true when increasing the total tree height h by another subtree of height h/d, whereby the total tree construction time can be estimated by the time required for the construction of a single subtree times d.

In conclusion a lower number of subtrees results in better worst-case signing times and should therefore be preferred if the memory specifications of the card allow so and longer

		Number of Subtrees				
		2	**3**	**4**	**5**	**6**
h/d	**2**	2:15	3:23	4:30	5:38	6:45
	3	4:30	6:45	9:00	11:15	13:30
	4	11:36	17:24	23:12	29:08	34:48
	5	23:48	35:42	46:08	59:30	71:24
	6	46:48	70:12	93:36	117:00	140:24
	7	93:36	140:24	187:12	234:00	280:48
	8	187:12	280:48	374:24	468:00	561:36

Table 5.6: Estimated initialization times for different tree sizes and subtree height/number ratios in minutes:seconds. These values are only estimates based on measurements. Inter-/Extrapolation was performed for values not directly measured. For tree sizes of up to $h = 20$ these values held true with a deviation of $< 10\%$ for all performed measurements

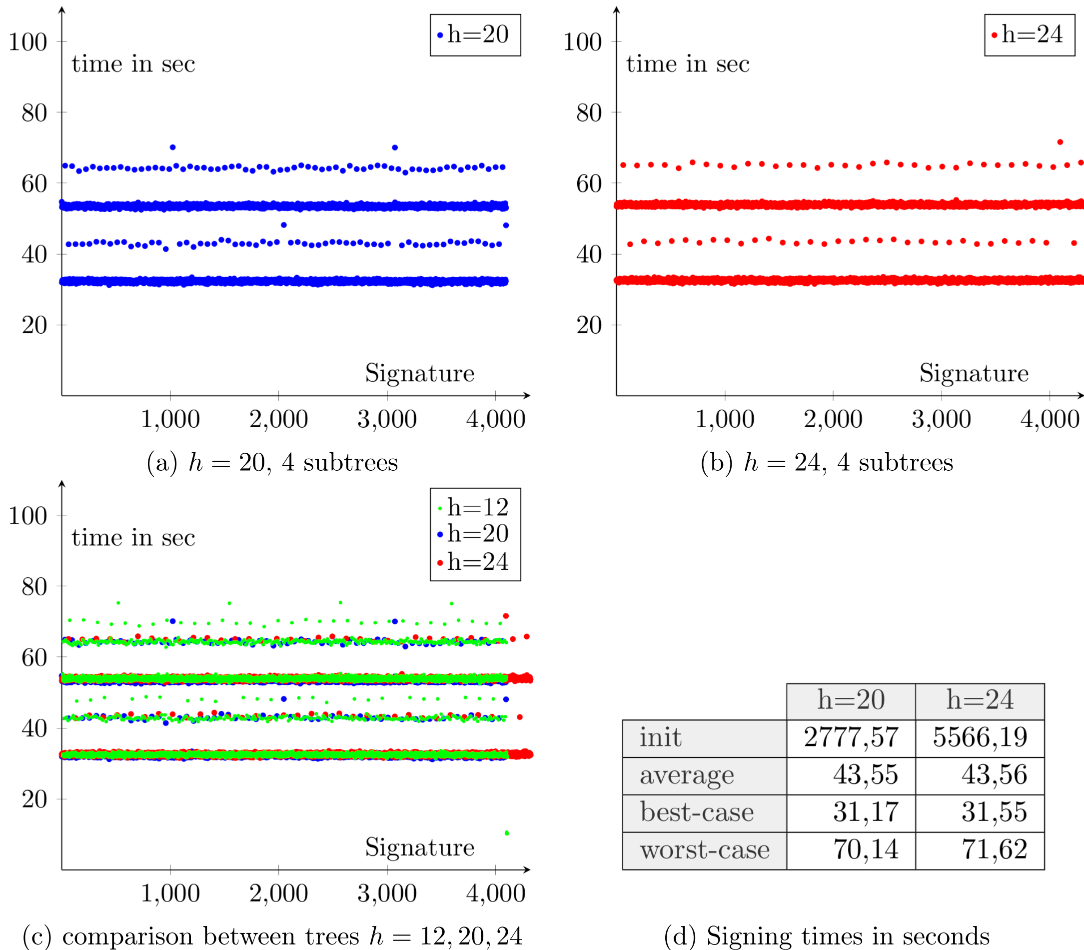
initialization can be accepted.

5.4 Test of larger trees

In order to get an understanding of the real-world capabilities of the implementation, tree structures capable of generating a number of signatures high enough for real-world applicability are now evaluated. To this end, trees of height $h = 20$ and $h = 24$ where chosen, thus allowing $2^{20} = 1.048.576$ and $2^{24} = 16.777.216$ signatures in total. Whether these sizes are sufficient for actual usage will be discussed in Chapter 6. Following from the results gathered in the previous section, the trees where constructed with the following parameters:

- $h = 20, h = 26$
- $w = 4$
- $d = 4$
- $k = h/d$

This time exhausting the whole trees was not possible due to the significantly larger amount of signatures. For the tree of height $h = 20$ again 4096 signatures where created. For the tree of height $h = 6$ a total of 4322 signatures where calculated. The additional signing operations for $h = 24$ where performed in order to see whether the signature times change after the first tree switch at signature 4095. As no deviation from the previous ones was found, the test was stopped after signature 4322. Again the signatures where tree switches where performed can clearly be seen for both cases in Figure 5.3a every 1024 signatures and during the preparation for the 4096th round for $h24$ in Figure 5.3b. Note that these are not the worst case times to be expected in these schemes, as only trees on the second layer have been switched. Worst case times are to be expected every $2 * 2^{(d-1)*h/(d)}$ signatures with the first one occurring at signature $2^{(d-1)*h/(d)} - 1$. At this round, the preparation of the authentication path for the next signature requires an BDS update on every subtree level. Figure 5.3c shows a comparison of the signature times of three different trees with each one



(a) $h = 20$, 4 subtrees

(b) $h = 24$, 4 subtrees

(c) comparison between trees $h = 12, 20, 24$

	h=20	h=24
init	2777,57	5566,19
average	43,55	43,56
best-case	31,17	31,55
worst-case	70,14	71,62

(d) Signing times in seconds

Figure 5.3: Signature generation times for trees of size $h = 20$ and $h = 24$, both with $d = 4$ subtrees. Subfigure 5.3c shows a comparison to the tree with parameter $h = 12$ $d = 4$ from Figure 5.2

.

consisting of $d = 4$ subtrees. Here the observation made in Section 5.3.3, that the size of the subtree only has a minor impact on the performance, is further confirmed. This allows the estimation that the worst-case signing time of the trees of height $h = 20$ and $h = 24$ is roughly the same as the one measured for $h = 12$ at 75.40 seconds. Note that this is only an estimation. In a real-world application there might be factors that influence the actual performance like degradation of the EEPROM or the CPU over time. The total memory requirement for storing the BDS states of the tree of height $h = 24$ amount to 16,5 KB. All in all the memory consumption of the applet running the tree of height $h = 24$ is 1,51 KB RAM and 50,53 KB EEPROM.

5.5 Conclusion of the Evaluation

In this section the different parameters influencing the signature times and sizes where compared. The parameter sets considered to be best suited for real-world usage where then used

on larger trees in Section 5.4. The results show that single signatures can be created in ≈ 11 seconds. This is however only possible if an authentication path is already prepared on the card. If a series of signatures is required or no authentication path was prepared when a signature is requested, the signing operation takes ≈ 44 seconds on average. The results also show that the cost of using larger trees is only an increased storage requirement with signature times not being affected.

Limits of further optimizations

The implementation presented here can likely be optimized in several places. However, some subroutines, especially the WOTS+ chaining function, requires a large amount of hash function calls by design, independent of the implementation. For example, consider the parameter set chosen in Section 5.4 is used. In this case a single WOTS+ pk generation requires 1197 calls to the hash function, each time with $3 * n$ byte input. This calculation is not only necessary at least every second signing round but must also be considered a worst case runtime for a signing operation as discussed in Subsection 4.5.1. Executing this number of hash function calls requires ≈ 4.84 seconds on the card. For the authentication path computation further signatures are required for the L-tree construction and the BDS update, writing operations to the EEPROM are necessary in order to store the next authentication path and numerous other subroutines must be performed. This allows the conclusion that even if a perfectly optimized implementation existed, signature times would still be several seconds long.

6 Analysis of possible use cases

Following from the findings presented in the previous chapter, an actual usability in the current state is unlikely, as runtimes far exceed those of classical digital signature schemes like RSA or ECDSA. However, the currently achievable runtimes could suffice in a number of cases. In the following chapter possible use-cases will be discussed.

6.1 Criteria for use-case analysis

In order to find possible use-cases for XMSS running on JavaCards, its limitations have to be taken into consideration.

6.1.1 Number of possible signatures

XMSS is a stateful scheme with a fixed number of signatures. When it is used as a signature scheme, this fact has to be taken into account as different use-cases require different numbers of signatures over a certain timespan. Requirements range from having to create ≈ 40 signatures per day when solely used for signing e-mails [1] to potentially thousands of signatures per second when used by a TLS server. By choosing the right tree size, the total number of possible signatures can be adjusted. However, this is only possible to a certain degree and especially limited when a SmartCard is being used.

6.1.2 Signature Size

Conventional signature schemes like RSA and (EC)DSA in most cases have signature sizes of less than 2048 bit [Sma]. Although hash based signatures introduce larger signatures, for some use-cases this might not be a limiting factor. With typical sizes of a single email in the range of 1,5 kbyte [OW14] to several Megabytes when data is attached, users and e-mail providers are accustomed to handling those data sizes. In other cases like when used in a VPN context, the increased signature size introduces new challenges. If for example VPN based on TLS is used, the standard only defines signatures with sizes of up to 4096 bytes [RD08]. Signatures of larger size increase the cost of the authentication process.

6.1.3 Runtime

Another factor is the signature generation time. Conventional signature schemes like RSA require less than 10 ms per signing operation of a hash output [BC13], therefore minimizing negative impact on the user experience. As the results presented in chapter 5 show, this is a factor to consider for XMSS running on JavaCards. Depending on the use-case, long runtimes can not only be annoying for the user but also critical for actual viability. For example if a

[1] https://info.templafy.com/blog/how-many-emails-are-sent-every-day-top-email-statistics-your-business-needs-to-know, Website visited on 11th of July 2020

signature for an e-mail is created, there is no strict runtime limit from a technical perspective. If the scheme is however used for authentication in a TLS handshake, timeouts can occur [2].

6.1.4 Security

SmartCards offer a number of security anchors as outlined in Subsection 2.9.6. For this reason they are used in conjunction with conventional signature schemes in many cases. Some companies or institutions require long time keys to be stored on hardware [EPC20] or require two-factor authentication where a SmartCard can be used [GCHW15]. The security provided by the JavaCard infrastructure also applies to the implemented scheme as no sk is exposed to the host computer and signatures are generated isolated on the card. Use-cases where SmartCards are already used are therefore also possible applications for JavaCard based XMSS as no security decline is to be expected.

6.1.5 Usability

In contrast to the aforementioned properties, usability is a less technical but also important factor to consider. Security should not be implemented with the cost of decreased usability, as this hinders the spread of security systems and even encourages users to circumvent them [BKS13]. The usage of a SmartCard and XMSS should therefore not have a negative impact on the user experience. Factors here are the requirement of specific hardware and software together with their interaction. Conn [Con95] describes the concept of time tolerance windows. These are the amount of time a user is willing to wait until he or she decides that a task will likely not complete. Fiona Nah [Nah04] analyzed the tolerance of users in the context of loading websites, coming to the conclusion that a majority of people are only willing to wait for $\approx$ two seconds. Lallemand et al.[LG12] focused on user's satisfaction when working with systems that impose wait times. They came to the conclusion that a previously constituted waiting time of 10 seconds [Nie93] is a threshold where users rated the waiting time disproportionate in relation to the result. Following from these results a wait time that is perceivable but less than 10 seconds is considered suboptimal but in a reasonable range. Anything over this time is consequently considered too long.

6.2 E-mail signing

The digital signing of E-Mails is an important countermeasure against phishing attacks, business email compromise and spoofing. Digital signatures allow the recipient to ensure the authenticity and integrity of the message, thus counteracting these threats. The most widely used standards for email signing are S/MIME [TR10] and PGP[FDC+07]. Early versions of PGP used RSA with the factorization problem as its basis while newer versions use the Elgamal algorithm that relies on the hardness of the DLP. S/MIME uses DSA for signing, which also relies on the DLP. As a result, none of these standards are secure against Shor's algorithm, given a sufficiently large QC existed.

[2] https://openvpn.net/faq/tls-error-tls-key-negotiation-failed-to-occur-within-60-seconds-check-your-network-connectivity/, Website visited 14th of July 2020

6.2.1 Current state

In a recent study Braun et al. [BO19] analyzed the usage of PGP by conducting a survey with 3727 participants. One of the questions asked was the reason why former users gave up on using PGP for e-mail encryption and signing. With 72.4 percent, the most common answer was that they had no communication partner that also used encryption. This shows the lack of dissemination of cryptographic systems in use when it comes to e-mail communication. Ruoti et al. [RAH^{+}16] researched the usability of secure e-mail communication. Their experiments showed that users prefer simple plug-in implementations. Increased complexity and hurdles in day-to-day usage are factors that reduce the desire for people to use cryptographic systems, even though they are interested in secure communication. This shows that usability is an important factor when wide spread acceptance of cryptographic schemes is desired.

6.2.2 Requirement Definition

In general, a novel signature scheme should not introduce drawbacks that were not present in established schemes as this would hinder wide-spread acceptance. For this reason, the already established schemes, S/MIME and PGP and the signature schemes they are based on are used as a reference. The following functional requirements will be defined and compared against the capabilities provided by the JavaCard based implementation.

Number of signatures

In order to find a number of signatures that XMSS must be able to provide, a realistic number of signatures generated by a RSA or DSA key pair must be found. For PGP, Google Payments states that an RSA private key should have a lifetime of no longer than two years [3]. This is however defined with respect to a payment application and following Murphy's Law, users can be expected to define an infinite lifetime. This means that in a worst-case scenario, the scheme is expected to be able to generate signatures over timespan that is practically not limited. Besides time, the second factor influencing the required number of signatures is the quantity of requests to be expected. The average number of e-mails written from a business account per day is 40 [4]. The study does not state whether only working days are counted into this statistic, but even when considering a 7-day week, the signatures that could be created by a tree of height $h = 20$ would suffice for ≈ 72. Note that this is a very rough estimation, with data based on the years 2014 - 2018. Since no deviating data was found, 40 emails per day will for now be considered an average requirement. This means that with a tree of this size the number of possible signatures should be sufficient for the working life of an average user. For this reason no negative impact on usability is to be expected.

Signature Size

As already stated in Section 6.1.2, signature sizes for XMSS are considerably larger than the ones users are used to from schemes like RSA and (EC)DSA. The actual size of the signature depends on the tree size and the parameters used. A tree of height $h = 26$ with a parameter set of $d = 4$ subtrees and $w = 4$ produces signature sizes of $\approx 17,4$KB. With a lower number

[3] https://developers.google.com/standard-payments/reference/best-practices, visited 13th of July 2020

[4] https://info.templafy.com/blog/how-many-emails-are-sent-every-day-top-email-statistics-your-business-needs-to-know, visited on 11th of July 2020

Figure 6.1: Client based signing: The message is transferred to the SmartCard and the generated Signature is returned

of subtrees this size can be further reduced. In comparison to e-mail attachments this is a relatively small amount of data.

Runtime

While not being a limiting factor from a technical perspective, signing times do affect the usability of the scheme in the real world. With respect to the goal of not negatively impacting the usability, the operation should not be perceivable by the user but at least within the time tolerance window defined in Section 6.1.5

6.2.3 Client vs. Server based signing

When creating digital signatures for e-mails in general, there are two approaches. Client based signing, where a digital signature is created directly on the user's computer, and server based signing, where signatures are issued by a key- or mail server. Both come with different hardware and software requirements and offer different kinds of advantages and disadvantages. Neither of the approaches is limited to a specific signature scheme or hardware to be used.

6.2.4 Client based signing

For this use-case consider a person that works in a company that requires hardware based signature generation and decided to introduce quantum secure signatures. The user works on a laptop either from inside the company's office or from home. It is further assumed that the IT department handled the installation of all required software and the distribution of the public key. In the following section, the requirements and the arising benefits will be discussed. figure 6.1 visualizes the client based signing approach.

Hardware Requirements

As the signatures are directly created by the user, he or she must be in possession of a SmartCard. Consequently, also a card reader is required, either built into the laptop or connected via USB.

Software Implementation

For a software implementation there are several possible approaches. While a stand alone program that is used to generate a signature independently from the used mail program is possible, an easier approach would be the usage of a plugin. From a software perspective the communication with the SmartCard could be handled by an extension to an existing e-mail client. This also follows results presented by Ruoti et al. [RAH+16], that this approach is preferred by users. When the user presses the send button, the plugin generates a hash value of the e-mail, transforms it into an APDU and sends it to the SmartCard over the PC/SC interface. After the signing operation is finished, the signature is received by the plug-in nested inside the response APDU, gets extracted and then automatically attached to the message. Then the actual e-mail transmission is started by the client. If two-step signing is used, the plugin must also trigger the preparation of the next authentication path.

Required user interactions

Compared to sending an e-mail without a signature, this approach requires the user to perform some additional steps. First of all, the SmartCard needs to be inserted into the reader. If the SmartCard is protected by a PIN, the user is then required to enter the key.

Security properties

As the SmartCard is directly accessed from the user's computer, the message is not transferred over a potentially insecure network before the signature is created. As a result, the communication security solely relies on the security provided by the PC/SC interface. As the writer of the e-mail uses an individual SmartCard with a specific pk, a recipient can verify the authenticity down to the single user, thus creating an end-to-end scheme. Note that this is not strictly required for being a countermeasure against common attacks like phishing, spoofing and business e-mail compromise, as authentication down to the company level is adequate. However in combination with a trusted instance this allows this scheme to fulfill the property of non-repudiation down to this person [OLZ08].

Requirement fulfillment and usability aspect

As discussed in Section 6.2.2, neither the number of signatures nor the signature size should be limiting factors when this scheme is used by a single person. If signatures are created with the two-step approach, signing times are less than a second longer than the requirements defined in Section 6.2.2. This fast signing time is only possible if the preparation of the next authentication path is finished before the next signature is required. If the preparation step is triggered immediately after the signature generation is finished, this time amounts to ≈ 34 seconds on average. Whether users frequently send e-mails in such a high frequency is to be doubted. Reliable data on this subject could not be found. It can however be assumed that users send an addendum to their previous message or send the message again to another

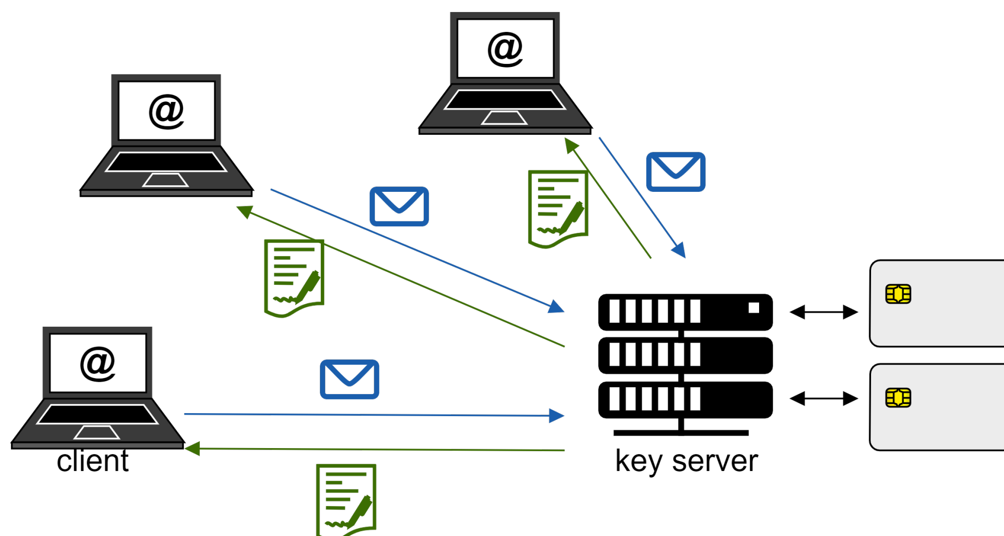


Figure 6.2: Key-server signing: The client sends a message to a key-server. The SmartCard attached to the server calculates a signature, which is then transferred back to the client

assignee they initially forgot in such a time frame. For this case, the additional time must be taken into account. In any case, the times required for signature generation are longer than defined in Section 6.2.2 and perceivable by the user.

An option to counteract this problem would be to hide the email transfer simply by performing the signing task in background and starting the transmission after completion. This is possible as no active user interaction is required after the send button has been pressed. However, for real-world usage this is problematic in a case where the user wants to send an e-mail shortly before shutting down the computer. In this case either the shutdown procedure needs to be postponed until the signing operation is complete or the transmission will be completed with the next startup. Both cases are such major limitations that they preclude use in a real-world environment.

A benefit of this client based approach is the ability to generate signatures without requiring an active internet connection. The user could compose messages while in the field, sign them using the SmartCard and queue the e-mail for sending until a network connection is re-established.

6.2.5 Server based signing

For the use case where signatures are created by a server, there are two possible approaches. One approach the usage of a key server, where the message is transferred to the key server, gets signed there and a signature is returned. Then the message with the attached signature is sent to the mail server by the signing computer. figure 6.2 visualizes the workflow of this approach. With the other approach, the server itself forwards the message after attaching the signature as visualized in figure 6.3. In both cases a Domain key [KCH11] or group signature scheme [MNF+12] can be implemented. In the now proposed use case consider

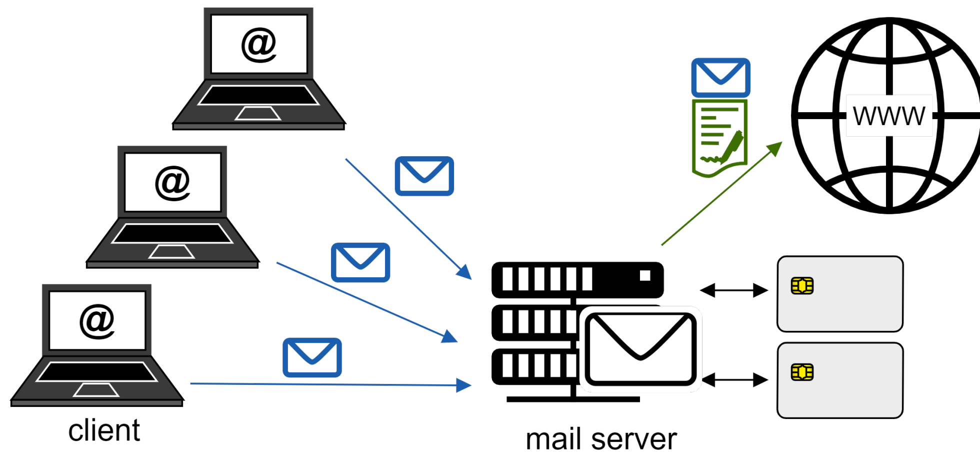


Figure 6.3: Mail server signing: The client sends a message to a mail server. The SmartCard attached to the server calculates a signature, which is attached to the e-mail and forwarded directly

company A with a self-hosted mail server running POSTFIX [5]. A new customer, company B, demands the usage of quantum secure signatures for all e-mails that A's employees send to B. B also requires A to store keys on a secure hardware module. Budget restrictions prohibit the acquisition of a HSM and providing every employee that might has to communicate with B with an individual SmartCard is also considered to costly. Company A therefore chooses to implement a server based signature scheme with signing operations performed on a SmartCard connected to their mail server.

Hardware Requirements

From the company's perspective the required hardware is minimal with a SmartCard and reader connected to their mail server. From an employee point of view no additional hardware is required.

Software Requirements

From an employees perspective no special software is required, as outgoing emails can be transmitted to their company's mail server like before the introduction of XMSS. The complexity of this scheme lies within the software side of the mail server. Here the communication with the SmartCard is handled by the server's software. For many e-mail agents, including POSTFIX, similar schemes using classical signature algorithms already exist[6]. This way a plug-in could be installed that handles the signing process. E-Mails addressed to company B can be signed by calculating their hash value, sending it to the SmartCard and attaching the signature encapsulated in the response APDU to the message before sending it to the destination server. Whether only e-mails addressed to specific addresses or all outgoing traffic

[5]https://de.postfix.org
[6]https://kofler.info/dkim-konfiguration-fuer-postfix/, visited 13th of July

should be signed could be adjusted in the plug-in configuration.

Required user interactions

As again all the complexity of the signing operation is handled by the mail server, no additional user interaction is required in this case.

Security properties

By not having individual signature per user but a common public key for a group of people, authentication of the sender is only possible down to a company or group level. A end-to-end signature scheme like it is possible with client based signing is not achieved using the proposed scheme. This still is adequate for acting as a countermeasure for common attacks like phishing and spoofing. Access to the server and consequently the signing operation is restricted to a group of people. A potential security risk compared to client based signing is introduced by the transmission between the user and company A's mail server. Here a different signature scheme needs to be present to ensure the integrity of the message when it arrives at the server.

Fulfillment of the functional requirements

Whether the number of signatures a single Card is able to provide is enough in this use-case depends on the number of users and the amount of e-mails written by them. In a worst-case scenario the company decides that all outgoing messages should be signed using this scheme. When again considering a single employee writes 40 signatures per day and and a key should have a life cycle of 2 years, a tree of height $h = 24$ offers enough signatures for approximately

$$\frac{2^{24}/40 \text{ signatures}}{2 * 260 \text{ working days}} \approx 806 \text{ employees}$$

While this might seem like a large number, the constraining factor here is not the amount of signatures but the time required to compute them. Even when considering a full day of 24 hours, the card is only able to compute ≈ 1964 signatures in this time, which is in turn only sufficient for ≈ 49 employees. Even if e-mail transmission is not required to complete immediately, the thereby introduced delays are not acceptable from a practical perspective. Estimating an actual maximum number of employees per card based on the signing times is hard, as it is likely that e-mails are not sent perfectly distributed over a working day. A more realistic estimation must therefore be significantly lower than 49 employees.

A way to increase the number of signatures the scheme is able to create in a certain time is the usage of multiple cards in as single server. The server could then distribute incoming requests to those cards, thus enabling parallel signing operation. This way also the number of signatures the scheme is able to create increases. This does however mean that more than one *pk* is used and all of them need to be distributed. A verifier then needs to check if one of these match their calculated *pk*. A different approach only requiring one global *pk* will be discussed later.

The signature size of messages as seen by the recipients is equal to the one described in the client based signing approach, while senders do not have to attach and transmit any signature bytes. Negative impact on the usability is therefore minimized when using this approach.

6.2.6 Key-server based signing

Like briefly mentioned in the beginning of Subsection 6.2.5, a sort of middle-ground between strictly client and server based approach is the usage of a key server. By using this approach users are also not required to possess a SmartCard. A client side plugin similar to the one discussed in Subsection 6.2.4 could be used to communicate with a key-server instead of a SmartCard. The plug-in transmits the e-mail to the key server and receives a signature that was calculated by a SmartCard connected to the key-server. The signature is then attached to the e-mail and sent outwards by the client. This approach again requires the client to wait for the signing operation to finish. As far as signature times are concerned, this approach is especially problematic as the worst traits of client and server based signing are combined here. The signature generation time cannot be hidden from the user, as the mail is sent by the client computer and the mail program therefore has to wait for the signature generation to finish. Furthermore the number of users per card must again be considered the same way as discussed for the server based approach. From a security perspective the key-server approach is similar to server based signing with the exception of a improvement concerning the message integrity. As the signature is sent back to the e-mail client plugin, the XMSS signature can be verified against the original message before sending. Therefore the integrity of the message transmission between the client and the key-server is also ensured by the XMSS signature on a quantum-secure level.

6.2.7 Conclusion for the e-mail use-case

The time required for signature generation is the main restraining factor when it comes to the real-world practicability of JavaCard based XMSS in the e-mail use-case. This applies for the client and the two server based approaches. In comparison to signing times of established schemes like S/MIME, a significant speedup is necessary in order to provide the same level of usability. Especially for client and key-server based signing, a runtime of no more than 10 seconds, preferably under 2 seconds is required. If those times would be achievable, the proposed systems would offer adequate performance for real-world usage. The number of signatures and their sizes are within the requirements and the server based approaches offer scalability through the usage of multiple cards.

6.3 Authentication during VPN key negotiation

With an increasing number of people working from home, establishing a secure connection to the internal network of their employer is often required. Virtual Private Networks enable this task by providing a secure tunnel over an insecure network. Digital Signatures are an important building block of this system, as they are commonly used for authentication in this context.

The exact requirements depend on the VPN protocol used. For example, OpenVPN[7] uses Transport Layer Security[RD08] while some security products like genuscreen[8] are based IPsec [FK11]. A paper by Butin et al. [BWB17] already implemented XMSS into TLS 1.2 for OpenSSL and analyzed real-world constraints. However, they did not analyze the VPN case but focused on HTTP over TLS (HTTPS). They argue that in this case, larger certificate

[7] https://openvpn.net/faq/why-openvpn-uses-tls/
[8] https://www.genua.de/loesungen/firewall-vpn-appliance-genuscreen.html

sizes are not a constraining factor as the average website in 2017 was already 2 megabytes[9]. For the following analysis, the focus will be laid on IPsec when used for a VPN.

6.3.1 Digital signatures in IPsec authentication

IPsec uses Internet Key Exchange Version 2 (IKEv2) [KHN+14] for key exchange and management. This protocol handles both the key exchange for symmetric encryption and authentication through digital signatures over an insecure network. It also manages the negotiation of the cryptographic schemes and agreement of key sizes used for them.

6.3.2 IKEv2 handshake

In IKEv2, the connection is established in two steps. The first step initializes the communication and exchanges key material for a Diffie-Hellman key exchange. In the second step the communication partner authenticate themselves to each other. The following section gives a short description of this process.

Initialization

The communication is initiated by one communication partner (initiator) who sends an IKE_SA_INIT UDP packet to another partner (responder). This first packet contains proposed security associations (SA) in the form of proposed message authentication algorithms, pseudorandom functions for key generation, encryption algorithm and a Diffie-Hellman group information. The responder answers to this packet by sending the accepted SAs. As both messages also include a key for a Diffie-Hellman key exchange, this allows the construction of a shared secret. This way the subsequent packages can be transmitted symmetrically encrypted.

Authentication

The next set of exchanged packets is where the communication partners authenticate each other. The Initiator sends a frame called IKE_AUTH with a corresponding frame answered by the responder. These frames are used to authenticate the previous messages, exchange certificates and establish a so-called Child SA, the actual SA used for the encrypted data transmission. This second pair of exchanged messages is the part where classical signature schemes are used to authenticate the communication parties.

Optional Reauthentication

These two steps are performed again when a reauthentication is initiated by the Initiator in order to generate a new set of SAs with new symmetric keys for encryption. In contrast to IKEv1, where a reauthentication is required every time the session keys are updated, IKEv2 supports rekeying without interrupting the connection. Simple rekeying does not require the partner to authenticate themselves again.

[9] https://httparchive.org/reports/page-weight, visited 27th of July

6.3.3 Requirements for a signature scheme in IKEv2

In order to be usable for IKEv2 authentication and consequently a VPN use-case, a set of requirements need to be fulfilled. For each of both communication partners a signature is required for the establishment of the connection by the IKEv2 handshake. In addition another authentication operation is performed whenever a reauthentication is performed. As IKEv2 allows rekeying without reauthentication, a digital signature is not always required if new Child SAs are negotiated. However, ensuring that the communication partners are still able to provide the required credentials is an important security factor. Otherwise the connection could be kept alive, even if a certificate is expired or one partner is no longer in possession of a required SmartCard [StS].

Number of Signatures

For both communication partners a signature is required for the establishment of the connection and for reauthentication, as in both cases a full IKEv2 handshake is performed. The StrongSwan default configuration is set to a rekey and reauthentication frequency of 4 hours [StS], while Cisco Meraki and Oracles VPN services based on IPsec have a default lifetime of 8 hours for their IKEv2 SAs [cis, Ora19].

Signature Size

The official specifications [KHN+14] require every IKEv2 implementation to be able to handle messages of lengths up to 1280 octets. This is a limit introduced by the IPv6 MTU. However, IKEv2 allows the IKE_AUTH packet to be fragmented into multiple UDP frames[Smy14]. This feature can only be used if both communication partners support it. The initiator suggests message fragmentation in the IKE_INIT packet and the responder has to agree to its usage. If both parties support it, no maximum IKE_AUTH package size is defined. Otherwise the packets should not succeed a size of 1280 octets.

Signature time

The IKEv2 RFC [KHN+14] does not state recommended timeout values, however it suggests that messages should be retransmitted at least a dozen times over a period of at least several minutes before a connection is terminated. Finding a definitive limit the signing operation is allowed to take is therefore not possible as it differs between actual implementations and is chosen based on the specific use-case or practical experience. A critical factor when it comes to the signature generation time is the way a reauthentication is performed. There are two ways, Break-before-make, the default case in StrongSwan, and Make-before-break. When the former way is used, the SA is first deleted and then the connection is reestablished, resulting in an interruption of the connection. Make-before-break does not introduce an interruption, as it first negotiates a new SA before replacing the old one with the new one. However, this is only possible if both communication partners are capable of handling overlapping SAs. For a complete evaluation both cases need to be considered.

Hardware and Software requirements

For hardware based authentication both communication partners naturally require the respective hardware, in the considered context a SmartCard with a reader. From a software

perspective the IPsec and IKEv2 implementations require support for the authentication method, in the context of this work XMSS, as they have to be able to agree to this scheme during the initial SA and Child SA negotiations.

6.3.4 Description and analysis of the use-case

For a real-world use-case, consider the situation where a company allows its employees to work from home. In order to access the internal network, a secure connection must be established to the remote computers, where a VPN solution based on StrongSwan is chosen. Users are required to perform a two factor authentication. Besides their password, users authenticate themselves with an individual SmartCard. For secure signature generation on the side of the company, a SmartCard is also used for signing operations in the VPN gateway. This gateway is the common entry point for all VPN connections, thus managing all connections to the remote computers. Since the company is a pioneer in security technology, they want the authentication process to be quantum secure. For this reason XMSS is deployed on all SmartCards.

Requirement fulfillment

As outlined in Section 6.3.3, the signature size should not exceed 1280 bytes unless message fragmentation is supported. This size is already exceeded by a single WOTS+ signature alone, regardless of which of the recommended Winternitz parameters is used. Message fragmentation must therefore be supported by the IKEv2 implementation in order to work with XMSS. As far as the number of signatures is concerned, the highest default reauthentication frequency of any VPN service found was 4 hours in case of StrongSwan. Considering a work day of 8 - 10 hours, this would amount to only one initial authentication and two reauthentications. This is a significantly lower number than the one discussed in the e-mail use case, where the number of possible signatures was shown to be sufficiently high. On the gateway side the number of authentications necessary depends on the number of users. When further considering the number of 4 authentications per user and day and a certificate with a lifetime of 10 years like it is the default for OpenVPN servers[10], the number of employees that could use a single card holding 2^{20} signatures could be used for

$$\frac{2^{24}/4 \text{ signatures}}{10 * 260\text{working days}} \approx 1613 \text{ employees}$$

Just like in Subsection 6.2.5, not the number of signatures but the time required to calculate them is the restraining factor. For the gateway side, the runtimes of signing operations are a major problem. It has to be assumed that connection requests are not perfectly distributed over a working day as the majority of the employees are likely to establish a connection when they begin to work in the morning. However, a card is only able to issue a signature every ≈ 44 seconds with the proposed parameters. If connection requests occur in a higher frequency, this would result in a queue building up, resulting in waiting times or even timeouts. For each user connecting during a signing operation, the time required to finish the queue increases by another 45 seconds. Estimating a maximum number of users per SmartCard is again hardly possible. Parallelizing authentication operations could again be achieved by using multiple cards.

[10] https://openvpn.net/faq/why-are-vpn-certificates-valid-for-10-years/

Whether signature times must be considered a critical factor for the remote communication partner depends on the VPN configuration. In the proposed use-case the company has the option to use Make-before-break when negotiating new Child SAs. In this case the connection does not suffer from interruptions prolonged by the time the SmartCard requires for signature generation. Following from the long timeout tolerances of the IKEv2 protocol mentioned in Section 6.3.3, even worst-case signing times as described in Section 5.4 are short enough to prevent a connection termination. Should Break-before-make be required by one of the communication partners, the signing times are a more critical factor as they introduce interruptions of the communication.

From a usability perspective the runtime becomes even more problematic due to the fact, that a connection is only established after two signing operations. Even if authentication paths are precomputed and no queue is present on the gateway, this introduces a waiting time of at least two times ≈ 11 seconds, thereby far exceeding the time tolerance window defined in Section 6.1.5. Using a SmartCard for signature generation on the gateway side is therefore not practicable. A limited practicability for JavaCard based XMSS on the client side can only be achieved if the authentication on the gateway is performed on more powerful hardware.

6.3.5 Conclusion for the VPN use-case

The exact requirements and therefore practicability of JavaCard based XMSS depends on the VPN's configuration with the most important factors being the support for message fragmentation and the method of reauthentication. When full control over the configuration of both communication partners is given, the scheme would technically be applicable for authentication from the perspective of a single user. From a usability perspective the same cannot be said, as the signing time again exceeds the time tolerance window defined in Section 6.1.5.

For usage within a gateway, the runtimes also become a problem from a functional perspective. In order to be applicable in this context, the scheme has to be able to allow the establishment of connections within the time tolerance window even if all users connect simultaneously. However, this is not achievable at the moment.

6.4 Summary

Following from the results presented in this chapter, the following section is intended to give an overview over the gathered results. Table 6.1 gives a visual overview of the results presented in the previous section. The common most restraining factor among all use-cases is the long runtime. This is especially problematic when signatures are required in a higher frequency than the ≈ 44 seconds the implementation requires on average per signature generation. Optimizing the implementation to an extent, where signature times are no longer a limiting factor is not likely as discussed in Section 5.5. In the end the performance required for fast execution of XMSS is not available on currently available JavaCards.

As far as the number of signatures and the signature size are concerned, the scheme is already able to fulfill most requirements. The number of signatures is estimated to be sufficiently large for the analyzed use-cases. The problem in the VPN use-cases concerning the limited compatibility if one communication partner does not support message fragmentation is not limited to a JavaCard based implementation but to XMSS in general. This is likely not

		Number	Runtime	Size	Usability	Overall
	Implemented	2^{24}	11/45 seconds	17,8KB		
E-mail	**Requirements**	40 per user & day	<10 sec			
	Client based	green	yellow	green	yellow	yellow
	Server based	green	red	green	yellow	red
	Key-Server	green	red	green	red	red
VPN	**Requirements**	4-5 per user & day	<10 sec	<1280 bytes		
	Client side	green	yellow	yellow	yellow	yellow
	Gateway side	green	red	yellow	red	red

Table 6.1: Visualization of the evaluation criteria and the estimated practicability. Green indicates a fulfillment of the requirement, yellow a limited an red an insufficient result. The overall result presented in the furthest right column is defined as the worst result in any single requirement of the specific use-case

the only case where the transition from conventional cryptographic schemes to post quantum ones requires adjustments in protocols and software. A quantum secure key exchange protocol called Intermediate Exchange [Smy20] also requires key fragmentation.

The usability of the scheme is influenced by multiple factors. This rating is intended to indicate the extent to which the user experience is negatively affected. As this is not a functional requirement and multiple factors influence the overall usability of the implementation in every use-case, a definitive statement is hard to make in some cases. Some runtimes can possibly be hidden from the user, as it is the case in server-based e-mail signing. Although the signing time in this case is considered insufficient, the overall usability was assessed at least limited. The reason behind this is the fact that here no additional interaction or hardware is required on the user's end. In addition the runtime can be hidden to a greater extend than it is possible in the key-server or client based approach. The longer signing times expected in comparison to client based signing is therefore considered to have less impact on the user experience. In the other cases the signing times are harder to hide, thus impacting the user experience more directly. Therefore the usability is more limited here and was therefore classified as insufficient.

The overall result gives an estimation on the general practicability of the implemented scheme. Here the functional requirements as well as the non-functional categories have to be considered. For a real-world applicability all of the single requirements need to be fulfilled. Therefore the result here is defined as the worst result in any single requirement of the specific use-case. All in all the cases where a single card per user is deployed are at least viable to a limited degree. In the other cases the achieved runtimes where simply not fast enough.

7 Conclusion and Future Work

7.1 Conclusion

The technical advancements in the field of quantum computing come at a cost. The results presented in this thesis show this first hand. The introduced computational effort necessary for creating XMSS signatures is significantly higher compared to conventional signature schemes like RSA or DSA. While technically capable of issuing XMSS Signatures, the hardware is not well suited for the task. The time required to create a single signature is longer than acceptable for most use-cases and achieving runtimes that are short enough is likely not possible on currently available JavaCards.

The increased computational cost is not the only challenge introduced by a transition to post-quantum cryptography with adjustments also necessary outside the signature schemes themselves. The increased signature sizes require design changes to protocols and in software. An example for this is the IKEv2 key exchange. While conventional schemes do not strictly require the support of message fragmentation, this feature is necessary when XMSS is used.

The XMSS^{MT} scheme implemented and analyzed in this thesis showed to be of limited practicability for real world usage. While usage within a multi-user environment cannot be considered viable, for a single client the scheme could be applicable. If only single signing operations are required, the scheme is able to issue signatures in ≈ 11 seconds. This is made possible by decoupling a majority of the total computational effort into a task that can be prepared in the background. During the actual signing operation only the minimal required subroutines must be performed. Also the number of signatures has proven not to be a limiting factor. Even with the limited memory available on card, trees of a sufficient size can be constructed. Furthermore, the overall tree size showed to have minimal impact on the overall and worst-case signing times.

Running XMSS^{MT} on JavaCards is an approach that has received little attention so far. The only other work dedicated to this problem by Laan et al. [LPR+18] achieved signing times significantly slower than the ones presented in this work. It should however be noted that a direct comparison between the two approaches is not possible due to the differences in Hardware. The problem they encountered regarding the worst case signing times whenever new subtrees are computed car however be mitigated in the implementation presented in this thesis through the usage of the BDS tree traversal algorithm.

7.2 Future Work

For real-world usage, a number of improvements would need to be made. The following section is intended to provide some ideas for solving these problems and presents topics for future research.

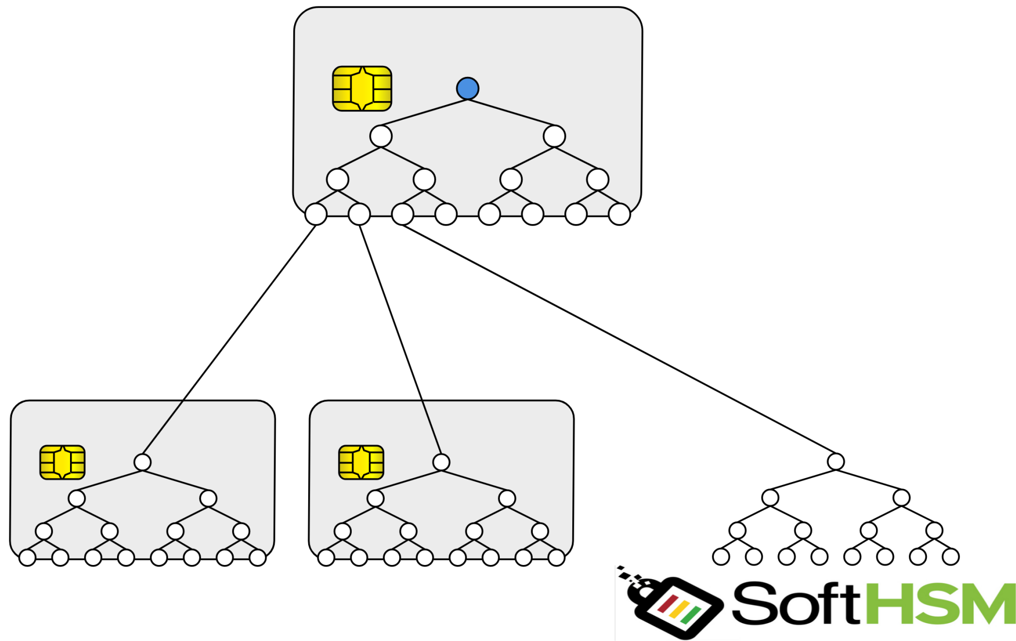


Figure 7.1: Subtree Signing

7.2.1 Distributed XMSS leaf computation

In the current implementation the times required for the authentication path preparation deviate by almost 20 seconds from round to round. In cases where a subtree is exhausted, the deviation increases additionally. This problem is due to the unbalanced distribution of XMSS leaf node computation as discussed in Section 5.3.2. A possible improvement here would be to distribute XMSS leaf node computation over two rounds. The first round calculates half of the WOTS+ *pk*, the next round finishes the key and constructs the L-tree. This way more balanced signing times could be achieved. This would however require additional storage for the partially constructed *sk* and *pk*.

Subtree signing

As already mentioned in subsection 6.2.5, a problem arising through the usage of multiple cards in a server scenario is the increasing number of public keys. A possible approach to both speed up the signature generation runtime as well as the number of possible signatures is establishing a scheme using layers of XMSS schemes. This construct consists of one or more individual XMSS schemes, each with its own *pk*. Those *pk* are then signed by another XMSS scheme, thus creating a scheme similar to regular XMSS^{MT} but distributed over separate instances. Such a scheme was introduced by McGrew et al. [MKF+16], who suggested an hybrid approach consisting of a stateless scheme on the top layer that is used to sign the *pk*s of stateful signature schemes which in turn are used for the actual signing operations. Figure 7.1 shows a visualization of such a scheme.

Using multiple cards

One example for an implementation of such a scheme would be the usage of multiple cards. In this case, consider the server holds $x > 1$ cards with an initialized XMSS tree. The pk of one of those cards, from now on referred to as the parent card, is now considered the global pk. This parent card is then used to sign the roots of the child cards, thereby creating a full XMSS$^{\mathrm{MT}}$ signature for every one of those cards. The task of the child cards is performing the actual signing operations requested by the users. They also create standard XMSS$^{\mathrm{MT}}$ signatures but do not return them immediately. Instead the before mentioned signature created by the parent card for this child is attached to this signature. This way two authentication paths are combined into one. Note here that the height h of the XMSS$^{\mathrm{MT}}$ trees does only have to be equal between the child cards. The tree of the parent card can be of any size as long as the subtree height is equal to the ones of the child cards. While this approach does not speed up the runtime of a signature generation on a single card, this scheme allows the usage of many cards in parallel, thus enabling a distribution of signing operations in a multi-user scenario. This way also the number of signatures that can be created is increased. By combining the authentication paths of parent and child card, the effective tree height is the sum of their individual heights. On the negative side this also increases the signature size. Consequently, the height of the XMSS instances on the cards could also potentially be decreased, thus allowing faster initialization times and lower memory requirements. For example in a case where 2^{20} signatures are required, XMSS trees of only height $h = 10$ on both the parent card as well as all the children would be sufficient to create the same number of signatures. However, as this would also require 2^{10} child cards, a smaller parent tree and larger child trees would be a more sensible choice. Furthermore, a smaller tree on the parent card results in a smaller signature that needs to be stored on the child card.

Using softHSM in combination with a SmartCard

The described scheme is not restricted to the use on SmartCards. As an alternative approach the scheme could also be realized using SoftHSM [1], a software based implementation of a Hardware Security Module, in place of solely using SmartCards on the child level. The global pk is still generated on the parent card. The XMSS schemes on the child level are however executed inside the SoftHSM environment, thus allowing faster signing operations as the performance of the host computer running the system can be used for signature generation. A consequence of this approach is a reduced security level, as the sks on the child level are no longer stored on a secure hardware module but inside software. If a child tree got compromised, a partial key revocation would need to be performed. This means that the compromised scheme's authentication on the parent tree gets declared invalid, therefore ensuring that new signatures of this child are no longer valid. As a result an updated version of the global pk would need to be distributed, containing the whole old pk and additionally a list of blacklisted authentication paths. This approach may seem impractical, as this not only requires a redistribution of the pk but also increases its size significantly, but the benefit here is, that the schemes on the child level remain their validity. Whether this approach is viable from a practical perspective could be a topic for future research.

[1] https://www.opendnssec.org/softhsm/

Nomenclature

pk	public key
sk	secret key
APDU	Application Protocol Data Unit
API	Application Programming Interface
BDS	tree traversal algorithm by Buchmann, Dahmen and Schneider
DSA	Dignital Signature Algorithm
ECDSA	Elliptic-Curve Digital Signature Algorithm
EEPROM	Electrically Erasable Read-Only Memory
HSM	Hardware Security Module
IKEv2	Internet Key Exchange Version 2
IP	Internet Protocol
IPsec	Internet Protocol security
JCOS	JavaCard Operating System
MTU	Maximum Transmission Unit
OTS	One-time signature
PQC	Post-quantum cryptography
QC	Quantum Computer
RAM	Random Access Memory
ROM	Read-Only Memory
RSA	Cryptographic algorithm by Rivest, Shamir and Adleman
SC	SmartCard
softHSM	Software based Hardware Security Module
UDP	User Datagram Protocol
VM	Virtual Machine

VPN	Virtual Private Network
WOTS+	Winternitz one-time signature scheme
XMSS	eXtended Merkle signature scheme
XMSS^{MT}	multi-tree eXtended Merkle signature scheme

List of Figures

2.1 L-tree construction visualized: Segments of the uncompressed WOTS+ pk and intermediate results are denoted as seg_x^y with x being the position on that L-tree layer and y being the layer within the tree 14
2.2 Authentication Path visualized 15
2.3 Signature structures of XMSS and XMSS^{MT} 16
2.4 Initialization of the BDS algorithm. [BDS08] 20
2.5 Typical SmartCard architecture 21
2.6 IFD-Subsystem as shown in [IFD05] 22
2.7 State diagram as shown in [LPR^{+}18]. Dottet arrows indicate APDU communication. Command APDUs triggering the single state computations are not shown. 28

4.1 Visualization of a signing operation with authentication path preparation. The received message is handed over to the signing function ①. Here the WOTS+ signature is created ② and the authentication path computed in the precious round is appended to it ③. Before the now complete XMSS signature is returned, the authentication path for the next round is computed ④ and stored im memory for the next round ⑤. Then the signing funktion returns the XMSS signature ⑥. It is then forwarded to a response function ⑦, which either returns an APDU containing the signature or sends its size if extended APDUs are not supported. 36

5.1 Signing times for different parameters k in the first 70 signatures of an XMSS^{MT} tree of height $h = 20$ with $d = 5$ subtrees 44
5.2 Comparison of three different ratios between subtree height and number. All trees are of height $h = 12$. Subfigure (d) with indicators for runtimes introduced by the respective subtree layer. 45
5.3 Signature generation times for trees of size $h = 20$ and $h = 24$, both with $d = 4$ subtrees. Subfigure 5.3c shows a comparison to the tree with parameter $h = 12$ $d = 4$ from Figure 5.2 48

6.1 Client based signing: The message is transferred to the SmartCard and the generated Signature is returned 54
6.2 Key-server signing: The client sends a message to a key-server. The SmartCard attached to the server calculates a signature, which is then transferred back to the client 56
6.3 Mail server signing: The client sends a message to a mail server. The SmartCard attached to the server calculates a signature, which is attached to the e-mail and forwarded directly 57

7.1 Subtree Signing 66

Bibliography

[AAB+19] Arute, Frank ; Arya, Kunal ; Babbush, Ryan ; Bacon, Dave ; Bardin, Joseph ; Barends, Rami ; Biswas, Rupak ; Boixo, Sergio ; Brandao, Fernando ; Buell, David ; Burkett, Brian ; Chen, Yu ; Chen, Jimmy ; Chiaro, Ben ; Collins, Roberto ; Courtney, William ; Dunsworth, Andrew ; Farhi, Edward ; Foxen, Brooks ; Fowler, Austin ; Gidney, Craig M. ; Giustina, Marissa ; Graff, Rob ; Guerin, Keith ; Habegger, Steve ; Harrigan, Matthew ; Hartmann, Michael ; Ho, Alan ; Hoffmann, Markus R. ; Huang, Trent ; Humble, Travis ; Isakov, Sergei ; Jeffrey, Evan ; Jiang, Zhang ; Kafri, Dvir ; Kechedzhi, Kostyantyn ; Kelly, Julian ; Klimov, Paul ; Knysh, Sergey ; Korotkov, Alexander ; Kostritsa, Fedor ; Landhuis, Dave ; Lindmark, Mike ; Lucero, Erik ; Lyakh, Dmitry ; Mandrà, Salvatore ; McClean, Jarrod R. ; McEwen, Matthew ; Megrant, Anthony ; Mi, Xiao ; Michielsen, Kristel ; Mohseni, Masoud ; Mutus, Josh ; Naaman, Ofer ; Neeley, Matthew ; Neill, Charles ; Niu, Murphy Y. ; Ostby, Eric ; Petukhov, Andre ; Platt, John ; Quintana, Chris ; Rieffel, Eleanor G. ; Roushan, Pedram ; Rubin, Nicholas ; Sank, Daniel ; Satzinger, Kevin J. ; Smelyanskiy, Vadim ; Sung, Kevin J. ; Trevithick, Matt ; Vainsencher, Amit ; Villalonga, Benjamin ; White, Ted ; Yao, Z. J. ; Yeh, Ping ; Zalcman, Adam ; Neven, Hartmut ; Martinis, John: Quantum Supremacy using a Programmable Superconducting Processor. In: *Nature* 574 (2019), 505–510. https://www.nature.com/articles/s41586-019-1666-5

[AAK19] Al-Odat, Z. ; Abbas, A. ; Khan, S. U.: Randomness Analyses of the Secure Hash Algorithms, SHA-1, SHA-2 and Modified SHA. In: *2019 International Conference on Frontiers of Information Technology (FIT)*, 2019, S. 316–3165

[AM96] Alfred Menezes, Scott V. Paul van Oorschot O. Paul van Oorschot: *Handbook of applied cryptography*. 1. CRC Press, 1996 (Discrete Mathematics and Its Applications). http://gen.lib.rus.ec/book/index.php?md5=F04B57C5F5DB1435C2ED3E439DA40BC0. – ISBN 9780849385230,0849385237

[AMG+16] Amy, Matthew ; Matteo, Olivia D. ; Gheorghiu, Vlad ; Mosca, Michele ; Parent, Alex ; Schanck, John: *Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3*. 2016

[BC13] *Kapitel* Advances in Communications, Computing, Networks and Security - Section 2 – Computer and Information Security. In: Bernabé, G. ; Clarke, N.: *Study of RSA Performance in Java Cards*. Bd. 10. 2013, S. 45–75

[BDH11] Buchmann, Johannes ; Dahmen, Erik ; Hülsing, Andreas: XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions.

In: Yang, Bo-Yin (Hrsg.): *Post-Quantum Cryptography*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–25405–5, S. 117–129

[BDS08] Buchmann, Johannes ; Dahmen, Erik ; Schneider, Michael: Merkle Tree Traversal Revisited. In: Buchmann, Johannes (Hrsg.) ; Ding, Jintai (Hrsg.): *Post-Quantum Cryptography*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. – ISBN 978–3–540–88403–3, S. 63–78

[Bea03] Beauregard, Stephane: Circuit for Shor's Algorithm Using 2n+3 Qubits. In: *Quantum Info. Comput.* 3 (2003), März, Nr. 2, S. 175–185. – ISSN 1533–7146

[BGD+06] Buchmann, Johannes ; García, Luis Carlos C. ; Dahmen, Erik ; Döring, Martin ; Klintsevich, Elena: CMSS – An Improved Merkle Signature Scheme. In: Barua, Rana (Hrsg.) ; Lange, Tanja (Hrsg.): *Progress in Cryptology - INDOCRYPT 2006*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. – ISBN 978–3–540–49769–1, S. 349–363

[BHT98] Brassard, Gilles ; HØyer, Peter ; Tapp, Alain: Quantum cryptanalysis of hash and claw-free functions. In: Lucchesi, Cláudio L. (Hrsg.) ; Moura, Arnaldo V. (Hrsg.): *LATIN'98: Theoretical Informatics*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. – ISBN 978–3–540–69715–2, S. 163–169

[BK09] Biryukov, Alex ; Khovratovich, Dmitry: Related-Key Cryptanalysis of the Full AES-192 and AES-256. In: Matsui, Mitsuru (Hrsg.): *Advances in Cryptology – ASIACRYPT 2009*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. – ISBN 978–3–642–10366–7, S. 1–18

[BKH14] Braun, J. ; Kiefer, F. ; Hülsing, A.T.: Revocation and non-repudiation : when the first destroys the latter. In: Katsikas, S. (Hrsg.) ; Agudo, I. (Hrsg.): *Public Key Infrastructures, Services and Applications (10th European Workshop, EuroPKI 2013, Egham, UK, September 12-13, 2013, Revised Selected Papers)*. Germany : Springer, 2014 (Lecture Notes in Computer Science). – ISBN 978–3–642–53996–1, S. 31–46

[BKS13] Blythe, J. ; Koppel, R. ; Smith, S. W.: Circumvention of Security: Good Users Do Bad Things. In: *IEEE Security Privacy* 11 (2013), Nr. 5, S. 80–83

[BO19] Braun, Sven ; Oostveen, Anne-Marie: Encryption for the masses? An analysis of PGP key usage. In: *Mediatization Studies* 2 (2019), 06, S. 69. `http://dx.doi.org/10.17951/ms.2018.2.69-84`. – DOI 10.17951/ms.2018.2.69–84

[BP] Bayern-PKI: *Was ist eine Smartcard?* https://www.pki.bayern.de/vpki/allg/sc/index.html,

[BSI20] *BSI TR-02102-1 "Kryptographische Verfahren: Empfehlungen und Schlüssellängen"*. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=12`. Version: März 2020

[Bus12] Busold, Christoph: *Hash-based Signatures on Smart Cards*, Technische Universität Darmstadt, mathesis, April 2012

[BWB17] Butin, D. ; Wälde, J. ; Buchmann, J.: Post-quantum authentication in OpenSSL with hash-based signatures. In: *2017 Tenth International Conference on Mobile Computing and Ubiquitous Network (ICMU)*, 2017, S. 1–6

[BZ08] Becker, Jesse ; Ziegler, Lydiy: *Design Considerations forImplementing EEPROM using theMC9S08DZ60*. Freescale Semiconductor Application Note. https://www.nxp.com/docs/en/application-note/AN3615.pdf. Version: 2008

[CDL+99] Cavallar, Stefania ; Dodson, Bruce ; Lenstra, Arjen ; Leyland, Paul ; Lioen, Walter ; Montgomery, Peter L. ; Murphy, Brian ; Riele, Herman te ; Zimmermann, Paul: Factorization of RSA-140 Using the Number Field Sieve. In: Lam, Kwok-Yan (Hrsg.) ; Okamoto, Eiji (Hrsg.) ; Xing, Chaoping (Hrsg.): *Advances in Cryptology - ASIACRYPT'99*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1999. – ISBN 978–3–540–48000–6, S. 195–207

[CDL+00] Cavallar, Stefania ; Dodson, Bruce ; Lenstra, Arjen K. ; Lioen, Walter ; Montgomery, Peter L. ; Murphy, Brian ; Riele, Herman te ; Aardal, Karen ; Gilchrist, Jeff ; Guillerm, Gérard ; Leyland, Paul ; Marchand, Jöel ; Morain, François ; Muffett, Alec ; Putnam, Chris ; Craig ; Zimmermann, Paul: Factorization of a 512-Bit RSA Modulus. In: Preneel, Bart (Hrsg.): *Advances in Cryptology — EUROCRYPT 2000*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000. – ISBN 978–3–540–45539–4, S. 1–18

[Che00] Chen, Zhiqun: *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. USA : Addison-Wesley Longman Publishing Co., Inc., 2000. – ISBN 0201703297

[cis] *Cisco Meraki Documentation - IPsec VPN Lifetimes*. https://documentation.meraki.com/MX/Site-to-site_VPN/IPsec_VPN_Lifetimes. – 16.07.2020

[CKRS20] Campos, Fabio ; Kohlstadt, Tim ; Reith, Steffen ; Stöttinger, Marc: LMS vs XMSS: Comparison of Stateful Hash-Based Signature Schemes on ARM Cortex-M4. In: Nitaj, Abderrahmane (Hrsg.) ; Youssef, Amr (Hrsg.): *Progress in Cryptology - AFRICACRYPT 2020*. Cham : Springer International Publishing, 2020. – ISBN 978–3–030–51938–4, S. 258–277

[CLP05] Coron, Jean S. ; Lefranc, David ; Poupard, Guillaume: A New Baby-Step Giant-Step Algorithm and Some Applications to Cryptanalysis. In: Rao, Josyula R. (Hrsg.) ; Sunar, Berk (Hrsg.): *Cryptographic Hardware and Embedded Systems – CHES 2005*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – ISBN 978–3–540–31940–5, S. 47–60

[Con95] Conn, Alex P.: Time Affordances: The Time Factor in Diagnostic Usability Heuristics. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. USA : ACM Press/Addison-Wesley Publishing Co., 1995 (CHI '95). – ISBN 0201847051, 186–193

[Cop97] Coppersmith, Don: Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. In: *J. Cryptology* 10 (1997), S. 233–260. http://dx.doi.org/10.1007/s001459900030. – DOI 10.1007/s001459900030

[Cor] Corporation, Oracle: *Java Card Platform 3.1 Development Kit User Guide - 14 Using Extended APDU*. 3.1. https://docs.oracle.com/en/java/javacard/3.1/guide/using-extended-apdu.html

[DDLM94] Denny, T. ; Dodson, B. ; Lenstra, A. K. ; Manasse, M. S.: On the factorization of RSA-120. In: Stinson, Douglas R. (Hrsg.): *Advances in Cryptology — CRYPTO' 93*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1994. – ISBN 978–3–540–48329–8, S. 166–174

[DK15] Delfs, Hans ; Knebl, Helmut: *Introduction to Cryptography - Principles and Applications*. 3. Springer-Verlag Berlin Heidelberg, 2015. http://dx.doi.org/10.1007/978-3-662-47974-2. http://dx.doi.org/10.1007/978-3-662-47974-2. – ISBN 978–3–662–47974–2

[DKL+18] Ducas, Léo ; Kiltz, Eike ; Lepoint, Tancrède ; Lyubashevsky, Vadim ; Schwabe, Peter ; Seiler, Gregor ; Stehlé, Damien: CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018 (2018), Feb., Nr. 1, 238-268. http://dx.doi.org/10.13154/tches.v2018.i1.238-268. – DOI 10.13154/tches.v2018.i1.238–268

[DS05] Ding, Jintai ; Schmidt, Dieter: Rainbow, a New Multivariable Polynomial Signature Scheme. In: Ioannidis, John (Hrsg.) ; Keromytis, Angelos (Hrsg.) ; Yung, Moti (Hrsg.): *Applied Cryptography and Network Security*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – ISBN 978–3–540–31542–1, S. 164–175

[Eat17] Eaton, Edward: *Leighton-Micali Hash-Based Signatures in the Quantum Random-Oracle Model*. Cryptology ePrint Archive, Report 2017/607, 2017. – https://eprint.iacr.org/2017/607

[Eke16] Ekerå, Martin: Modifying Shor's algorithm to compute short discrete logarithms. In: *IACR Cryptol. ePrint Arch.* 2016 (2016), S. 1128

[EPC20] *Guidelines on cryptographic algorithmsusage and key management*. European Payment Council. https://www.europeanpaymentscouncil.eu/sites/default/files/kb/file/2020-03/EPC342-08v9.0GuidelinesonCryptographicAlgorythmsUsageandKeyManagement_0.pdf. Version: März 2020

[Erd04] Erdmann, Monika: *Benchmarking von Java Cards*, Ludwig-Maximilians-Universität München, mathesis, Mai 2004

[FDC+07] Finney, Hal ; Donnerhacke, Lutz ; Callas, Jon ; Thayer, Rodney L. ; Shaw, David: *OpenPGP Message Format*. RFC 4880. http://dx.doi.org/10.17487/RFC4880. Version: November 2007 (Request for Comments)

[FK11] Frankel, Sheila ; Krishnan, Suresh: *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. RFC 6071. http://dx.doi.org/10.17487/RFC6071. Version: Februar 2011 (Request for Comments)

[GCHW15] Gabriel, Meghan ; Charles, Dustin ; Henry, JaWanna ; Wilkens, Tricia L.: *State and National Trends of Two-Factor Authentication for Non-Federal Acute Care Hospitals*. ONC Data Brief. https://www.healthit.gov/sites/default/files/briefs/oncdatabrief32_two-factor_authent_trends.pdf. Version: November 2015

[GE19] Gidney, Craig ; Ekerå, Martin: *How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits*. 2019

[Gro96] Grover, Lov K.: A Fast Quantum Mechanical Algorithm for Database Search. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. New York, NY, USA : Association for Computing Machinery, 1996 (STOC '96). – ISBN 0897917855, 212–219

[HBB13] Hülsing, Andreas ; Busold, Christoph ; Buchmann, Johannes: Forward Secure Signatures on Smart Cards. In: Knudsen, Lars R. (Hrsg.) ; Wu, Huapeng (Hrsg.): *Selected Areas in Cryptography*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–35999–6, S. 66–80

[HBG+18] Huelsing, Andreas ; Butin, Denis ; Gazdag, Stefan-Lukas ; Rijneveld, Joost ; Mohaisen, Aziz: *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. http://dx.doi.org/10.17487/RFC8391. Version: Mai 2018 (Request for Comments)

[Hia14] Hiary, Ghaith: A deterministic algorithm for integer factorization. In: *Mathematics of Computation* 85 (2014), 08. http://dx.doi.org/10.1090/mcom3037. – DOI 10.1090/mcom3037

[HRB13] Hülsing, A.T. ; Rausch, L. ; Buchmann, J.: Optimal parameters for XMSSMT. In: *Security engineering and intelligence informatics*. Germany : Springer, 2013 (Lecture Notes in Computer Science). – ISBN 978–3–642–40587–7, S. 194–208

[Hül13] Hülsing, Andreas: W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes. In: Youssef, Amr (Hrsg.) ; Nitaj, Abderrahmane (Hrsg.) ; Hassanien, Aboul E. (Hrsg.): *Progress in Cryptology – AFRICACRYPT 2013*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–38553–7, S. 173–188

[Hü13] Hülsing, Andreas: *Practical Forward Secure Signatures using Minimal Security Assumptions*, Technische Universität Darmstadt, phdthesis, August 2013

[IFD05] *Interoperability Specification for ICCs and Personal Computer Systems, Part 4. IFD Design Considerations and Reference Design Information*. http://pcscworkgroup.com/Download/Specifications/pcsc1-10_v2.01.14.zip. Version: September 2005

[ISO20] *ISO/IEC 7816-4:2020(en)*. https://www.iso.org/obp/ui/iso:std:iso-iec:7816:-4:ed-4:v1:en, 2020. – Identification cards — Integrated circuit cards — Part 4: Organization, security and commands for interchange

[KCH11] Kucherawy, Murray ; Crocker, Dave ; Hansen, Tony: *DomainKeys Identified Mail (DKIM) Signatures*. RFC 6376. http://dx.doi.org/10.17487/RFC6376. Version: September 2011 (Request for Comments)

[KGHGC19] Kranzlmüller, Dieter ; Guggemos, Tobias ; Höb, Maximilian ; Grundner-Culemann, Sophia: *Ein erster Überblick*. http://www.nm.ifi.lmu.de/teaching/Vorlesungen/2019ss/QuantumComputing/. Version: 2019. – Lecture Notes Einführung in Quantencomputing

[KHN+14] Kaufman, Charlie ; Hoffman, Paul E. ; Nir, Yoav ; Eronen, Pasi ; Kivinen, Tero: *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. http://dx.doi.org/10.17487/RFC7296. Version: Oktober 2014 (Request for Comments)

[KL07] Katz, Jonathan ; Lindell, Yehuda: *Introduction to Modern Cryptography (Chapman Hall/Crc Cryptography and Network Security Series)*. Chapman Hall/CRC, 2007. – ISBN 1584885513

[Kob94] Koblitz, Neal: *A Course in Number Theory and Cryptography*. 2. Springer-Verlag New York, 1994. http://dx.doi.org/10.1007/978-1-4419-8592-7. http://dx.doi.org/10.1007/978-1-4419-8592-7. – ISBN 978–1–4419–8592–7

[LG12] Lallemand, Carine ; Gronier, Guillaume: Enhancing User eXperience during waiting time in HCI: Contributions of cognitive psychology, 2012

[LPR+18] Laan, Ebo van d. ; Poll, Erik ; Rijneveld, Joost ; Ruiter, Joeri de ; Schwabe, Peter ; Verschuren, Jan: Is Java Card ready for hash-based signatures? In: *IACR Cryptol. ePrint Arch.* 2018 (2018), S. 611

[LYH11] Li, L. ; Yu, Z. ; Hao, Y.: Estimates of EEPROM device lifetime. In: *Tsinghua Science and Technology* 16 (2011), Nr. 2, S. 170–174

[MCF19] McGrew, David ; Curcio, Michael ; Fluhrer, Scott: *Leighton-Micali Hash-Based Signatures*. RFC 8554. http://dx.doi.org/10.17487/RFC8554. Version: April 2019 (Request for Comments)

[MD18] Maheswari, A.U. ; Durairaj, P.: Modified shanks' baby-step giant-step algorithm and Pohlig-Hellman algorithm. In: *International Journal of Pure and Applied Mathematics* 118 (2018), 01, S. 47–56. http://dx.doi.org/10.12732/ijpam.v118i10.48. – DOI 10.12732/ijpam.v118i10.48

[Mer90] Merkle, Ralph C.: A Certified Digital Signature. In: Brassard, Gilles (Hrsg.): *Advances in Cryptology — CRYPTO' 89 Proceedings*. New York, NY : Springer New York, 1990. – ISBN 978–0–387–34805–6, S. 218–238

[MG15] Mamaluy, Denis ; Gao, Xujiao: The fundamental downscaling limit of field effect transistors. In: *Applied Physics Letters* 106 (2015), Nr. 19, 193503. http://dx.doi.org/10.1063/1.4919871. – DOI 10.1063/1.4919871

[MKF+16] McGrew, David ; Kampanakis, Panos ; Fluhrer, Scott ; Gazdag, Stefan-Lukas ; Butin, Denis ; Buchmann, Johannes: State Management for Hash-Based Signatures, 2016. – ISBN 978–3–319–49099–1, S. 244–260

[MNF+12] Manulis, Mark ; Nils, Fleischhacker ; Felix, Günther ; Franziskus, Kiefer ; Bertram, Poettering: *Group Signatures: Authentication with Privacy.* BSI Publication. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/GruPA/GruPA.pdf?__blob=publicationFile. Version: August 2012

[MoT19] MoTechno: *Datasheet NXP.J3H145, NXP Semiconductors SmartMX2.* https://www.motechno.com/wp-content/uploads/J3H145-JCOP3.pdf. Version: August 2019

[Nah04] Nah, Fiona Fui-Hoon: A study on tolerable waiting time: how long are Web users willing to wait? In: *Behaviour & Information Technology* 23 (2004), Nr. 3, 153-163. http://dx.doi.org/10.1080/01449290410001669914. – DOI 10.1080/01449290410001669914

[Nie93] Nielsen, Jakob: *Usability Engineering.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993. – ISBN 0125184050

[OLZ08] Onieva, Jos A. ; Lopez, Javier ; Zhou, Jianying: *Secure Multi-Party Non-Repudiation Protocols and Applications.* 1st. Springer Publishing Company, Incorporated, 2008. – ISBN 0387756299

[Ora19] *IPSec VPN Best Practices - Oracle White Paper.* https://www.oracle.com/a/ocom/docs/ipsec-vpn-best-practices.pdf. Version: Mai 2019

[Ort03] Ortiz, C. E.: *An Introduction to Java Card Technology - Part 1.* https://www.oracle.com/technetwork/java/embedded/javacard/documentation/javacard1-139251.html, Mai 2003

[OW14] Oliviero, Andrew ; Woodward, Bill: *Cabling: The Complete Guide to Copper and Fiber-Optic Networking.* 5th. USA : SYBEX Inc., 2014. – ISBN 1118807324

[PAF19] Pierre-Alain Fouque, Paul Kirchner Vadim Lyubashevsky Thomas Pornin Thomas Prest Thomas Ricosset Gregor Seiler William Whyte Zhenfei Z. Jeffrey Hoffstein H. Jeffrey Hoffstein: *Falcon: Fast-Fourier Lattice-basedCompact Signatures over NTRU, Specifications v1.0.* https://falcon-sign.info/falcon.pdf. Version: 2019

[PK17] Panos Kampanakis, Scott F.: *LMS vs XMSS: Comparion of two Hash-Based Signature Standards.* Cryptology ePrint Archive, Report 2017/349, 2017. – https://eprint.iacr.org/2017/349

[Pol75] Pollard, J. M.: A monte carlo method for factorization. In: *BIT Numerical Mathematics* 15 (1975), Nr. 3, 331–334. https://doi.org/10.1007/BF01933667. – ISSN 1572–9125

[PZ03] Proos, John ; Zalka, Christof: Shor's Discrete Logarithm Quantum Algorithm for Elliptic Curves. In: *Quantum Information Computation* 3 (2003), 02

[RAH+16] Ruoti, Scott ; Andersen, Jeff ; Heidbrink, Scott ; O'Neill, Mark ; Vaziripour, Elham ; Wu, Justin ; Zappala, Daniel ; Seamons, Kent: "We're on the Same Page": A Usability Study of Secure Email Using Pairs of Novice Users. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (2016), May. `http://dx.doi.org/10.1145/2858036.2858400`. – DOI 10.1145/2858036.2858400. ISBN 9781450333627

[RD08] Rescorla, Eric ; Dierks, Tim: *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. `http://dx.doi.org/10.17487/RFC5246`. Version: August 2008 (Request for Comments)

[RE10] Rankl, Wolfgang ; Effing, Wolfgang: *Smart Card Handbook*. 4th. Wiley Publishing, 2010. – ISBN 0470743670

[RED+08] Rohde, Sebastian ; Eisenbarth, Thomas ; Dahmen, Erik ; Buchmann, Johannes ; Paar, Christof: Fast Hash-Based Signatures on Constrained Devices. In: Grimaud, Gilles (Hrsg.) ; Standaert, François-Xavier (Hrsg.): *Smart Card Research and Advanced Applications*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. – ISBN 978–3–540–85893–5, S. 104–117

[Reg04] Regev, Oded: *Lattices in Computer Science - Attack on RSA with Low Public Exponent*. `https://cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/rsa.pdf`. Version: 2004. – Lecture Notes

[RSA78] Rivest, R. L. ; Shamir, A. ; Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In: *Commun. ACM* 21 (1978), Februar, Nr. 2, 120–126. `http://dx.doi.org/10.1145/359340.359342`. – DOI 10.1145/359340.359342. – ISSN 0001–0782

[Sch19] Scherer, Wolfgang: *Mathematics of Quantum Computing*. 1. Springer-Verlag GmbH, 2019 `https://www.ebook.de/de/product/38295265/wolfgang_scherer_mathematics_of_quantum_computing.html`. – ISBN 978–3–030–12358–1

[Sho94] Shor, P. W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. USA : IEEE Computer Society, 1994 (SFCS '94). – ISBN 0818665807, 124–134

[Sho97] Shor, Peter W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. In: *SIAM Journal on Computing* 26 (1997), Oct, Nr. 5, 1484–1509. `http://dx.doi.org/10.1137/s0097539795293172`. – DOI 10.1137/s0097539795293172. – ISSN 1095–7111

[Sma] Smart, Nigel P. ; ECRYPT-CSA (Hrsg.): *Delivierable 5.4 Algorithms, Key Size and Protocols Report (2018): EU H2020-ICT Project Report*. `https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf`

[Sma15] SMART, Nigel P.: *Cryptography Made Simple*. 1st. Springer Publishing Company, Incorporated, 2015. – ISBN 3319219359

[Smy14] SMYSLOV, Valery: *Internet Key Exchange Protocol Version 2 (IKEv2) Message Fragmentation*. RFC 7383. `http://dx.doi.org/10.17487/RFC7383`. Version: November 2014 (Request for Comments)

[Smy20] SMYSLOV, Valery: Intermediate Exchange in the IKEv2 Protocol / Internet Engineering Task Force. Version: Juni 2020. `https://datatracker.ietf.org/doc/html/draft-ietf-ipsecme-ikev2-intermediate-04`. Internet Engineering Task Force, Juni 2020 (draft-ietf-ipsecme-ikev2-intermediate-04). – Internet-Draft. – Work in Progress

[StS] *strongSwan User Documentation - Expiry and Replacement of IKE and IPsec SAs*. `https://wiki.strongswan.org/projects/strongswan/wiki/ExpiryRekey`

[Szy04] SZYDLO, Michael: Merkle Tree Traversal in Log Space and Time. In: CACHIN, Christian (Hrsg.) ; CAMENISCH, Jan L. (Hrsg.): *Advances in Cryptology - EUROCRYPT 2004*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–24676–3, S. 541–554

[TR10] TURNER, Sean ; RAMSDELL, Blake C.: *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. RFC 5751. `http://dx.doi.org/10.17487/RFC5751`. Version: Januar 2010 (Request for Comments)

[Tun17] *Kapitel* 9 - Smart Card Security. In: TUNSTALL, Michael: *Smart Cards, Tokens, Security and Applications*. Berlin, Heidelberg : Springer, 2017. – ISBN 978–3–319–50500–8

[VCL+15] VALENTA, Luke ; COHNEY, Shaanan ; LIAO, Alex ; FRIED, Joshua ; BODDULURI, Satya ; HENINGER, Nadia: *Factoring as a Service*. Cryptology ePrint Archive, Report 2015/1000, 2015. – `https://eprint.iacr.org/2015/1000`