Hash-based Signatures on Smart Cards

Master-Thesis von Christoph Busold
April 2012

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Theoretische Informatik
Kryptographie und Computeralgebra

Hash-based Signatures on Smart Cards

Vorgelegte Master-Thesis von Christoph Busold

Betreuer: Prof. Dr. Johannes Buchmann und Andreas Hülsung

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 27. April 2012

(Christoph Busold)

Contents

1 Introduction

Digital signatures are one of the basic principles in public-key cryptography. They can be used to verify data from unsecure sources like the internet with the following purposes:

- **Authentication**: Ensure that a message in fact originates from the specified sender
- **Integrity**: Ensure that a message has not been modified in any way during transmission
- **Non-Repudiation**: Once a message is signed the author cannot deny having signed it at a later time

Digital signatures become especially interesting in interaction with smart cards, as they serve not only as secure storage for the private key, but are also able to generate signatures and keys themselves. This means the private key never leaves the protected environment of the smart card and cannot be accessed directly, even when the host computer is compromised.

Today, algorithms for digital signatures are mostly based on number theoretic problems like the Discrete Logarithm Problem or the Integer Factorization Problem. This includes the popular RSA scheme as well as the standardized Digital Signature Algorithm (DSA) and its Elliptic Curve variant ECDSA. Although those algorithms are generally considered secure, this cannot be proven reliably. Just because no promising solutions for their underlying problems have yet been found, does not mean they do not exist.

This changes suddenly as soon as one considers quantum computers. In 1994, the mathematician Peter Shor presented a quantum algorithm that is able to factor integers in polynomial time and hence easily break any RSA key. It can be found in [Sho96] together with a similar quantum computer solution for the Discrete Logarithm Problem.

So far, no one has succeeded in building a quantum computer large enough to be used for this purpose and it does not seem to be imminent. The threat did however motivate a new research area called Post-Quantum Cryptography for algorithms which remain secure in a world with quantum computers, including digital signature schemes.

One of the most promising candidates in this field are hash-based signatures introduced by Ralph Merkle in 1979. Their security relies solely on a cryptographic hash function. The Merkle Signature Scheme is flexible because the underlying hash function can be simply replaced with a different one. Up to now, there exist no attacks on arbitrary hash functions, even with quantum computers.

The intention of this thesis is to implement post-quantum signatures on a smart card using a modified version of Merkle called Extended Merkle Signature Scheme. In order to get fast run times, the hash function should make use of available hardware acceleration. Many smart cards feature dedicated coprocessors for cryptographic operations.

The extended Merkle scheme also comes with an interesting property called *Forward Security*. The idea of Forward Security for digital signature schemes was originally proposed in [And02] and later defined in [BM99]. It means that even if a signature key is compromised at some point in time, an attacker is not able to forge previously generated signatures. Therefore signatures issued before this point can remain valid. This requires some kind of order so that a verifier can distinguish between past and successive signatures. Schemes without this property need a time stamping authority that adds the time of the signature creation from the outside.

The following presents three use cases for digital signature schemes together with smart cards. Each of them has partly different requirements regarding their security level, signature and key sizes as well as run times.

Document Signing: One application is to sign documents like invoices, contracts or purchase orders. With digital signatures this process can be carried out via the internet as used for example in online banking. In some countries, digital signatures have a status equal to handwritten ones by law. In this case, non-repudiability becomes essential in order to make the signature legally binding.
This application requires a signature scheme with long-term security, as those documents often have to remain valid for decades. The size of a signature on the other hand is less restricted, because the documents themselves are typically an order of magnitude larger. Signature generation times should be usable for interactive processing.

Authentication: Authentication can be implemented with digital signatures using a challenge-response technique. The challenge is a random message which has to be signed by the authenticating party with its signature key. This method can be used to provide access control to computer systems and networks or physical locations like buildings and facilities using the smart card as digital identity card.

The requirements here are almost the opposite of those above: Signatures are created frequently, but their validity period is very short. Long-term security is of no importance as the signature scheme can be replaced easily before it becomes insecure. Generation times and sizes of signatures and keys should be low to reduce load and traffic. However, as signatures typically do not need to be stored, their size is less important.

Code Signing: Code signing is already quite common on personal computers, for example with operating system or anti-virus program updates. With the increasing complexity and usage of smart card software, code updates in order to fix bugs or add functionality become more and more important. For example, the 2010 bug in many German EC cards was fixed by updating the code at the terminals[1]. Without proper protection, such an update mechanism could be misused to inject malicious code and compromise the secure environment of the smart card.
In contrast to the previous applications, the signatures and keys are not generated on the card and their performance is therefore less significant. Instead, the card is equipped with the public key and only has to handle the verification. Signature and key sizes should be small, in order to reduce memory overhead.

Relating to these applications, this thesis focuses on document signing. Due to the long-term security requirements, post-quantum cryptography is more interesting in this case. Nevertheless, the other two will still be considered as possible applications.

The performance should be suitable for interactive use. Expressed in run times, this means that signature generation should be possible in less than one second. Otherwise the user would probably feel a delay in the process. Key generation is different, as it is typically done only occasionally. Waiting one minute here should be acceptable.

Unfortunately, the number of possible signatures with the Merkle scheme is limited and has a great influence on its performance, especially the time required for key generation. If we aim at one million signatures, this should be sufficient for document signing and even for authentication.

Regarding related work, there is already a previous implementation for a different version of the Merkle scheme on a smart card processor architecture by Rohde et al in [RED+08]. They assume that keys can be generated on a different system with far more computational power. Therefore they do not implement key generation on the smart card. Furthermore, their keys are limited to 65536 possible signatures.

This thesis is structured as follows: Chapter 2 explains the fundamentals of hash-based signatures, starting with the definition of hash functions over one-time signatures to the Merkle scheme and finally the concept of tree chaining. The following chapter 3 is about smart cards and their interface. Chapter 4 describes the implementation of the signature scheme. Its performance results are illustrated and analyzed in chapter 5, which also gives a brief evaluation of the side-channel resistance. Chapter 6 compares our results to those of Rohde et al and the performance of RSA and ECDSA on the same smart card. Furthermore, the security level of the presented scheme is discussed and compared to that of the others. Finally, chapter 7 gives a conclusion of this thesis and presents some topics for future work.

[1] See http://www.heise.de/security/meldung/Softwareupdate-fuer-Chips-koennte-EC-Karten-Problem-loesen-2-Update-897730.html

2 Hash-based Signatures

This chapter describes how digital signatures can be created out of hash functions. The two concepts used are Winternitz One-Time Signatures and the Merkle Signature Scheme.

The first section defines the primitives for both schemes: Pseudo-Random-Function family, (Forward Secure) Pseudo-Random-Generator and Hash Function.

Section 2.2 shows how one-time signatures are constructed with a Pseudo-Random-Function family using the Winternitz OTS. Winternitz only works with messages with a fixed size. In order to sign messages with arbitrary length, one can use the Hash-and-Sign paradigm with the help of a collision resistant hash function, see section 2.1.4. In this thesis, methods for signature generation and verification will always expect messages with correct length.

The next section 2.3 describes the Merkle Signature Scheme and its hash tree used to combine a fixed but arbitrary number of one-time keys. It is extended by a modified version in section 2.4, which lowers the requirements on the hash function and adds forward security to the signature scheme.

Finally Tree Chaining in section 2.5 allows to reduce the high initialization run times for a single Merkle tree by using multiple levels of trees. It also describes the disadvantages of this method, namely the increased size and more unbalanced generation time of signatures, and shows how their impact can be lowered by distributing computations amoung all signature steps.

Throughout this thesis, whenever the right shift operator >> is used, it is defined mathematically as

$$x >> y = \left\lfloor \frac{x}{2^y} \right\rfloor$$

and the left shift operator << analog as

$$x << y = x \cdot 2^y.$$

2.1 Primitives

The following four functions are the basic building blocks of both the Winternitz and the Merkle Signature Scheme.

2.1.1 Pseudo-Random-Function

A *Pseudo-Random-Function* (PRF) family is defined as

$$F_n = \{f_k : \{0,1\}^n \rightarrow \{0,1\}^n \mid k \in \{0,1\}^n\}$$

where the key k selects a specific function from the family. A PRF has the property that its output can not be distinguished from a function, which is chosen randomly from the set of all functions with the same input and output length.

2.1.2 Pseudo-Random-Generator

A *Pseudo-Random-Generator* (PRG) is a function

$$\text{prg} : \{0,1\}^n \rightarrow \{0,1\}^*$$

which uses an input seed $x \in \{0,1\}^n$ to create an arbitrary number of pseudo-random output bits.

A PRG can be implemented by applying the PRF f on the sequence of inputs $0, 1, 2, \ldots$ using the seed as its key. Accordingly, the output looks like

$$f_{seed}(0) \parallel f_{seed}(1) \parallel f_{seed}(2) \ \ldots$$

For a number of bits which is not a multiple of n, the last block could be truncated.

2.1.3 Forward Secure Pseudo-Random-Generator

In order to make pseudo-random data generation forward secure, a *Forward Secure Pseudo-Random-Generator* (FSPRG)

$$\begin{aligned} \text{fsprg}: \{0,1\}^n &\rightarrow \{0,1\}^n \times \{0,1\}^n \\ \text{SeedIn} &\rightarrow (\text{Rand}, \text{SeedOut}) \end{aligned}$$

is needed. It uses the Pseudo-Random-Generator described above to create two pseudo-random data blocks out of the input seed. One of them is returned as output value and the other is used to update the seed.

Forward security means that you can only compute the seeds in one direction, if you use a chain of FSPRG calls. Given that the old seeds are discarded or overwritten by the new ones, it is impossible to reconstruct previous seeds or output values.

2.1.4 Hash Function

The last one is the *Hash Function*. In cryptography, hash functions have three properties:

- **(First-)Preimage Resistance**: Given a hash output h, it is impossible to find any message m so that $h = \text{hash}(m)$, in other words it is one-way
- **Second-Preimage Resistance**: Given a message m_1, it is impossible to find a second one $m_2 \neq m_1$ with equal hash values $\text{hash}(m_1) = \text{hash}(m_2)$
- **Collision Resistance**: It is impossible to find any two different messages $m_1 \neq m_2$ with equal hash values $\text{hash}(m_1) = \text{hash}(m_2)$

With the Hash-and-Sign paradigm, a cryptographic hash function is used to create the initial message digest which is then signed. In this case, verifying a message means actually verifying its hash value. For this purpose, a collision resistant hash function is essential. Otherwise an attacker could find a different message with the same hash value and use it with the same signature.

This message digest hash function can be defined as

$$\text{md}: \{0,1\}^* \rightarrow \{0,1\}^m.$$

It computes a fixed size hash value out of an arbitrary length input message. This can be done with any secure hash function like SHA2 and is not implemented in this thesis.

Due to the birthday paradox, a hash function needs at least an output length of $2n$ for a security level of n bit against collision attacks, see section 8.3 in [GB08]. This means for 128-bit security m must be at least 256.

In section 2.3 the Merkle scheme needs a hash function to construct its hash tree. It always gets two tree nodes as input to compute one parent. Therefore it is explicitly defined with a fixed input length of two times the output like

$$\begin{aligned} \text{hash}: \{0,1\}^{2n} &\rightarrow \{0,1\}^n \\ \text{Left} \parallel \text{Right} &\rightarrow \text{Parent} \end{aligned}$$

where the left and right child nodes are concatenated.

2.2 Winternitz One-Time Signature Scheme

The Winternitz One-Time Signature Scheme (WOTS) was first described in [Mer89]. My implementation uses a slightly modified version from [BDE+11].

As described in this paper, the required one-way function is implemented with a PRF family where the input $x \in \{0,1\}^n$ is chosen randomly and always remains constant for one Winternitz instance. The notation $f_k^i(x)$ means the function is iterated i times using the output as key for the next stage, that is $f_k^2(x) = f_{f_k(x)}(x)$ and $f_k^0(x) = x$.

First we define the *Winternitz parameter* $w > 1$ which corresponds to the base of the message representation. Winternitz always signs messages with a fixed and predefined number of bits m. Thus a message with bit length m is represented in base-w as $M = (M_1, \ldots, M_{l_1})$ where the number of digits is

$$l_1 = \left\lceil \frac{m}{\log_2(w)} \right\rceil.$$

Additionally, a checksum

$$C = \sum_{i=1}^{l_1} (w - 1 - M_i)$$

is appended to the original message, again represented in base-w as $C = (C_1, \ldots, C_{l_2})$ where

$$l_2 = \left\lfloor \frac{\log_2(l_1(w-1))}{\log_2(w)} \right\rfloor + 1.$$

Therefore the total number of base-w digits is $l = l_1 + l_2$ and we define $(b_1, \ldots, b_l) = M \parallel C$.

The Winternitz secret key consists of l n-bit data blocks $(sk_1, \ldots, sk_l)$ chosen at random. The corresponding public key is computed by applying the one-way function $w - 1$ times onto each block of the secret key.

$$(pk_1, \ldots, pk_l) = (f_{sk_1}^{w-1}(x), \ldots, f_{sk_l}^{w-1}(x))$$

The signature of a message M is calculated as

$$(\sigma_1, \ldots, \sigma_l) = (f_{sk_1}^{b_1}(x), \ldots, f_{sk_l}^{b_l}(x)).$$

Verification means reconstructing the public key from the signature via

$$(pk_1, \ldots, pk_l) = (f_{\sigma_1}^{w-1-b_1}(x), \ldots, f_{\sigma_l}^{w-1-b_l}(x))$$

and comparing the result to the original public key.

The parameters defining one Winternitz instance are w, m, x plus the resulting lengths l_1, l_2 and l.

The checksum is necessary to prevent an attacker with a valid message and signature pair from easily forging a new pair by incrementing one of the message digits M_i and performing the required iterations on the corresponding data block. Incrementing any digit of the message will decrement the checksum, which requires the attacker to compute a preimage of one of the checksum blocks.

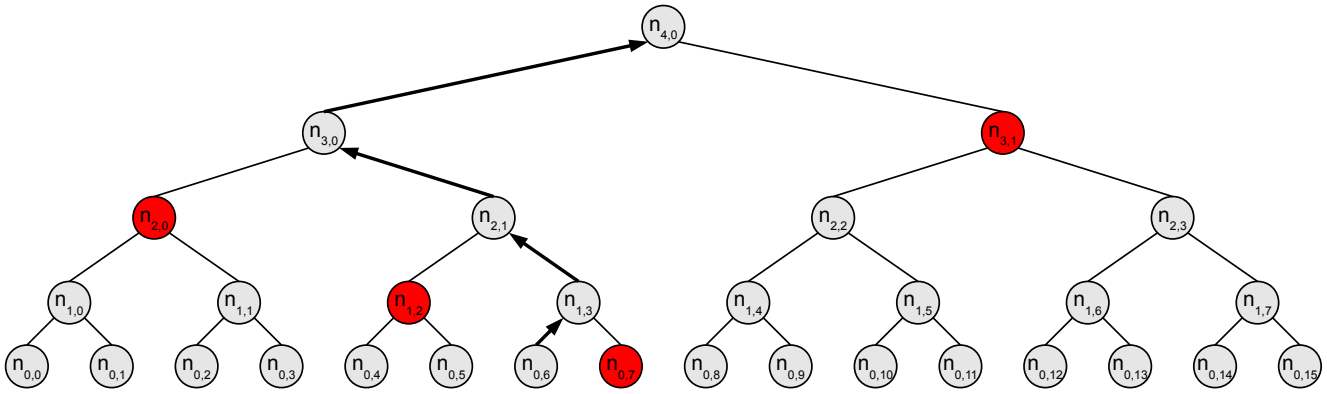


Figure 2.1: A Merkle tree with height $H = 4$. Every parent node corresponds to the hash output of the concatenation of its children. According to the convention, leaves start at height 0 while the root is at H. The red nodes mark the authentication path for leaf number 6.

2.3 Merkle Signature Scheme

The main disadvantage of one-time signatures is that you can use a key pair - as the name already suggests - only once. This can be solved with the *Merkle Signature Scheme* (MSS), which allows to bind multiple one-time signature key pairs to a single public key using a hash function. It was introduced by Ralph Merkle in [Mer89]. An overview of the complete scheme and description of the algorithms for tree hashing and authentication path computation can be found in [BBD08]. This also includes the extensions CMSS and GMSS explained later.

The Merkle scheme uses a binary hash tree of height H as shown in figure 2.1, where every leaf node corresponds to the hash value of a one-time public key. Per definition, leaves start at height 0 while the root, which serves as new public key, is at height H. The notation $n_{j,i}$ stands for the i-th node on height j. Parent nodes are computed by hashing the concatenation of their child nodes. The *Authentication Path* $(\text{Auth}_0, \ldots, \text{Auth}_{H-1})$ of a leaf consists of all sibling nodes on its way to the tree root. Figure 2.1 shows the authentication path for leaf 6 marked in red.

The Merkle signature for a leaf consists of its authentication path, leaf number, one-time public key and the one-time signature of the actual message. This gives a verifier all the information he needs in order to generate the corresponding leaf node, reconstruct the path to the root and compare it to the Merkle public key. This approach generally works with any one-time signature scheme. Winternitz has the advantage however, that the one-time public key can be reconstructed from the one-time signature and therefore does not need to be included explicitly in the Merkle signature.

With a tree height of H, the Merkle scheme is able to generate 2^H signatures, one with each one-time public key. However, saving all 2^H one-time keys is generally not possible. Therefore they should be generated on demand using a pseudo-random generator.

Limited space presents a similar problem during key generation, when the root for the public key needs to be computed. For this purpose, the treehash algorithm in section 2.3.2 is used to generate a tree (or subtree) iteratively with minimal memory requirements.

As every Merkle signature needs the authentication path for its corresponding leaf node, section 2.3.3 presents an algorithm to compute the next authentication path based on the current one.

The remaining sections describe the functions representing the signature scheme: KeyGen, Sign and Verify.

2.3.1 Seed Computation

The Winternitz secret keys are generated with a pseudo-random generator. This requires a different seed for each key, called one-time seed. The l data blocks are computed from the one-time seed like

$$(sk_{i,1}, \ldots, sk_{i,l}) \leftarrow \text{prg}(\text{otsSeed}_i)$$

where $i \in \{0, \ldots, 2^H - 1\}$.

Algorithm 2.1 The functions to initialize and update treehash instances

```
function TREEHASH.INITIALIZE(Start)
    CurrentLeaf ← Start
end function

function TREEHASH.UPDATE
    node_1 ← GenLeafNode(GenPK(CurrentLeaf))
    height ← 0
    while height = height of top node on Stack do
        node_2 ← Stack.pop()
        node_1 ← GenParentNode(node_1, node_2)
        height ← height + 1
    end while
    Stack.push(node_1)
    CurrentLeaf ← CurrentLeaf + 1
end function
```

These 2^H one-time seeds can in turn be computed out of a single starting seed using again the pseudo-random generator. According to the definition in section 2.1.2, the PRG uses a PRF with the initial seed as key and the input is counting upwards. Therefore the i-th one-time seed can be computed directly from its number as

$$\text{otsSeed}_i \leftarrow f_{\text{Seed}}(i).$$

In the following sections, a function GenPK, that generates the Winternitz public key for the i-th leaf, is supposed to be available:

$$(pk_{i,1}, \ldots, pk_{i,l}) \leftarrow \text{GenPK}(i)$$

2.3.2 Treehash Algorithm

The Treehash algorithm in 2.1 is used to construct the root of a hash tree with height H, using a stack with space for H nodes. It is initialized with a starting leaf number.

The idea is to construct one leaf in each step and successively compute its parent with nodes already on the stack. They are combined for as long as they have the same height, therefore the stack contains at most one node on each height at the same time. After 2^H steps the root is the only remaining node on the stack.

2.3.3 Authentication Path Generation

The authentication path for the next leaf $s+1$ is generated with algorithm 2.2 proposed in [BDS08] from the current algorithm state, including the previous authentication path.

First we define τ as the height of the first ancestor of the current leaf s, which is a left node. If s itself is a left node, then $\tau = 0$. Now the new authentication path for leaf $s+1$ always has to be updated on heights $0, \ldots, \tau$.

On height τ it shifts from a right node to its left sibling. If $\tau = 0$, then this new left authentication node corresponds to the current leaf and can be computed directly from its number s:

$$\text{Auth}_0 = \text{GenLeafNode}(\text{GenPK}(s))$$

For $\tau > 0$ the left node can be generated easily by saving the previous right authentication node on height $\tau - 1$ in Keep. The corresponding left sibling can be found in the current authentication path:

$$\text{Auth}_\tau = \text{GenParentNode}(\text{Auth}_{\tau-1}, \text{Keep}_{\tau-1})$$

Algorithm 2.2 Authentication Path Computation

```
function TREE.GENNEXTAUTH
    τ ← max{h : 2^h | (s+1)}
    if s >> (τ+1) is even and τ < H−1 then
        Keep_τ ← Auth_τ
    end if

    if τ = 0 then
        Auth_0 ← GenLeafNode(GenPK(s))
    else
        Auth_τ ← GenParentNode(Auth_{τ−1}, Keep_{τ−1})
        for h ← 0 to min{τ−1, H−K−1} do
            Auth_h ← Treehash_h.pop()          ▷ Access the node saved inside the instance
            if s+1+3·2^h < 2^H then
                Treehash_h.Initialize(s+1+3·2^h)
            else
                Treehash_h is not needed anymore
            end if
        end for
        h ← H−K
        while h < τ do
            Auth_h ← Retain_h.pop()
            h ← h+1
        end while
    end if
end function
```

Therefore Keep requires space for H nodes.

All heights below τ need new right nodes, which have to be precomputed from scratch. As this is more expensive the higher the node, it is advisable to save those close to the root during key generation.

For that reason we select a parameter $2 \leq K \leq H$, called *Retain height*, so that $H-K$ is even. Right nodes on heights h between $H-K$ and $H-1$ are saved in stacks Retain_h, except for the first one which is already part of the initial authentication path. This requires $2^{H-h-1}-1$ nodes on each height h.

Therefore the total number of retain nodes with retain height K is

$$\begin{aligned} & \sum_{i=H-K}^{H-1} (2^{H-i-1}-1) \\ = \; & (2^{K-1}-1) + \cdots + (2^0-1) \\ = \; & \sum_{i=0}^{K-1} (2^i-1) \\ = \; & \sum_{i=0}^{K-1} 2^i - K \\ = \; & 2^K - K - 1. \end{aligned} \tag{2.1}$$

On each height below $H-K$, a treehash instance computes upcoming right authentication nodes from the start using the treehash algorithm. In each step, a number of at most $(H-K)/2$ updates is distributed between those instances, so that they are finished by the time their next node is requested.

The instance to be updated is chosen each time by the lowest tail node. In case Treehash_h does not have a tail node, which means it is initialized but has not received any update yet, its height is defined as h. If there is more than one, those with lower heights are preferred. This concept is implemented in algorithm 2.3. Instances which are finished or uninitialized are not taken into account, meaning the set of instances that can be updated might be empty. In that case the loop is exited. In the end, the number of remaining updates is returned.

Algorithm 2.3 Distribution of updates between treehash instances of a Merkle tree

function TREE.UPDATE(budget)
  for $i \leftarrow 1$ **to** budget **do**
    $k \leftarrow \min\left\{h : \text{Treehash}_h.\text{height} = \min_{j=0,\ldots,H-K-1}\left\{\text{Treehash}_j.\text{height}\right\}\right\}$
    Treehash_k.Update()
  end for
  return number of unused updates
end function

Using this update algorithm, all treehash instances can share a single stack with a maximum of $H-K-2$ nodes, while saving one node in the instance itself. This works because at the time Treehash_h receives its first update, the lowest tail node of all higher instances is at least h, otherwise they would have been updated first. Therefore Treehash_h is finished before any of them receives another update. Similarly Treehash_h cannot interfere with lower instances. If Treehash_i with $i < h$ begins to store nodes on the stack, the lowest tail node of Treehash_h has to be higher than i. That means Treehash_i is finished before Treehash_h continues.

In order to satisfy the bound, Treehash_h stores only nodes up to height $h-2$ on the stack. The first node on height $h-1$ is saved in the instance and then combined with the second one to the final node on height h. The internal node can be initialized with the second right node on each height, saving some updates at the beginning.

After a node is taken from a treehash instance in algorithm 2.2, it is reinitialized to compute the next node starting with leaf number $s+1+3\cdot 2^h$, unless this is already outside the tree. In that case the instance is not needed anymore.

Altogether the algorithm state consists of

- the leaf number s and initial seed,
- the current authentication path with H nodes,
- Keep with H nodes,
- 2^K-K-1 retain nodes,
- $H-K$ treehash instances with one node and current leaf number each and
- a shared stack with $H-K-2$ nodes.

2.3.4 Key Generation

Key generation means computing the root of the tree and initializing the state along the way. A random initial seed is supposed to be available. The private key corresponds to the algorithm state defined above.

The authentication path starts with the first right node on each height, that is $\text{Auth} = (n_{0,1},\ldots,n_{H-1,1})$.

The retain stacks are filled with right nodes as

$$\text{Retain}_h.\text{push}(n_{h,2j+3})$$

for $h = H-K,\ldots,H-2$ and $j = 2^{H-h-1}-2,\ldots,0$.

The treehash instances are initialized with

$$\text{Treehash}_h.\text{push}(n_{h,3})$$

for $h = 0,\ldots,H-K-1$.

2.3.5 Signature Generation

The authentication path and leaf number for each signature can be taken from the current state. Then, the one-time seed is computed and the message is signed with the resulting Winternitz private key. This Winternitz signature is returned together with the current authentication path and leaf number as Merkle signature defined as $\sigma = (s,\ \text{Auth},\ \text{Sig}_{\text{ots}})$.

Thereafter the state is updated by incrementing the current leaf number, generating the authentication path for the next signature and distributing updates as described in section 2.3.3 above.

2.3.6 Verification

In order to verify a Merkle signature, first the Winternitz public key is reconstructed from the Winternitz signature and the message. Then the corresponding leaf node is generated as

$$n_0 \leftarrow \text{GenLeafNode}(\text{pk}_1, \ldots, \text{pk}_l)$$

and combined recursively with the authentication path using the leaf number s as

$$n_h \leftarrow \begin{cases} \text{GenParentNode}(\text{Auth}_{h-1},\ n_{h-1}) & \text{if } s >> (h-1) \text{ is odd} \\ \text{GenParentNode}(n_{h-1},\ \text{Auth}_{h-1}) & \text{if } s >> (h-1) \text{ is even} \end{cases}$$

for $h = 1, \ldots, H$. The signature is valid, iff the output n_H equals the root of the Merkle public key.

2.4 Extended Merkle Signature Scheme

The *Extended Merkle Signature Scheme* (XMSS) proposed in [BH11] differs from the original scheme in the generation of leaf and parent nodes. Additionally the computation of one-time seeds is modified in section 2.4.1 using the forward secure pseudo-random-generator. This adds forward security to the complete scheme. The necessary signature order follows directly from their numbering.

The idea to insert random bit masks into the tree construction was first described in [DOTC08]. With this extension it can be proven that the signature scheme is existentially unforgeable, provided that the used hash function is second-preimage resistant. This is a significant advantage over the original scheme, which required a collision resistant hash function for its security proof. As mentioned in section 2.1.4, the length of hash values for collision resistance must be twice as large as for second-preimage resistance.

The computation of parent nodes is modified as follows: For each height $j \in \{1, \ldots, H\}$ a bit mask $b_j \in \{0,1\}^{2n}$ is chosen at random. In order to generate a parent, its left and right child are concatenated, XORed bitwise with the corresponding mask and then passed to the hash function:

$$n_{j,i} = \text{hash}((n_{j-1,2i} \parallel n_{j-1,2i+1}) \oplus b_j)$$

Leaf nodes are also constructed differently: Instead of simply hashing the one-time public key, its l data blocks are used to build another hash tree similar to the Merkle tree. As l is usually not a power of two, the last left node on each height is lifted onto the next height if it has no right sibling. Parent nodes are generated analog to the Merkle tree but with different masks. This requires additional $\lceil \log_2 l \rceil$ bit masks used for all leaf nodes.

Algorithm 2.4 Construction of leaf nodes with XMSS

```
function GenLeafNode(pk_1, ..., pk_l)
    l* ← l
    while l* > 1 do
        t ← l* >> 1
        for i ← 1 to t do
            pk_i ← GenParentNode(pk_2i, pk_2i+1)
        end for
        if l* is odd then
            pk_t+1 ← pk_l*
        end if
        l* ← ⌈l*/2⌉
    end while
    return pk_1
end function
```

The root of this tree finally corresponds to the leaf node of the Merkle tree. Algorithm 2.4 shows how this concept can be implemented.

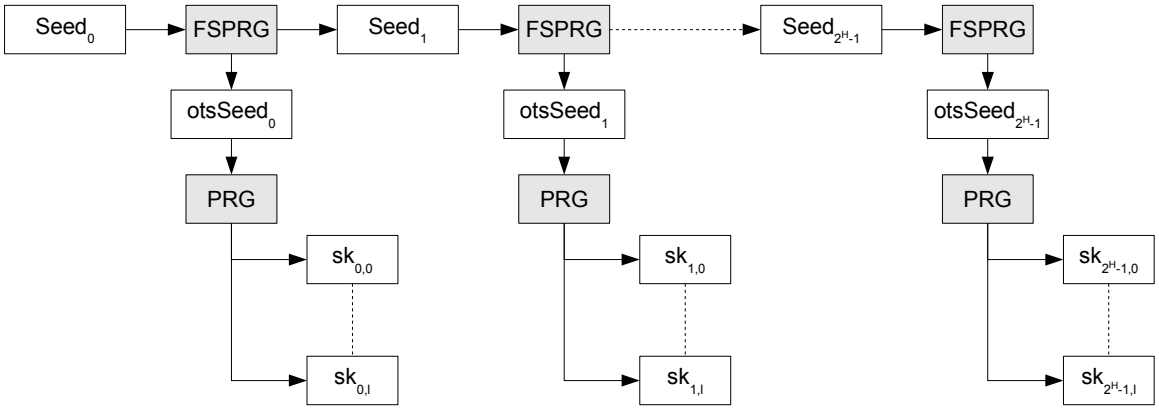


Figure 2.2: Forward secure computation of one-time keys and their seeds: The l data blocks for each of the 2^H Winternitz keys are still constructed with the PRG from a one-time seed. But the one-time seeds are now computed with a chain of FSPRG calls out of the starting seed on the far left.

In case the output size of the PRF s_{prf} does not match that of the hash function s_{hash}, the public key is interpreted as bit string, divided into

$$l^* \leftarrow \left\lceil \frac{l \cdot s_{\text{prf}}}{s_{\text{hash}}} \right\rceil$$

hash nodes and the last one is padded with zeros if necessary.

Altogether XMSS needs $\lceil \log_2 l \rceil + H$ bit masks, each with a size of $2n$ bit. These masks are placed in the public key, because they are needed for the generation of leaf and parent nodes again during verification. Therefore they should be unique and unpredictable for each key pair but do not need to be kept secret. This increases the size of the XMSS public key considerably.

2.4.1 Forward Secure Seed Computation

With the seed computation in section 2.3 a one-time seed can be generated directly from its number. This approach is simple but not forward secure. If you are in possession of the initial seed, you can reconstruct any previous one-time seed and Winternitz secret key.

The seed computation can be modified into a forward secure version using the forward secure pseudo-random generator from section 2.1.3. Starting from an initial seed as Seed_0, the current seed is updated in each signature step and the output value is used as one-time seed:

$$(\text{otsSeed}_i,\ \text{Seed}_{i+1}) \leftarrow \text{fsprg}(\text{Seed}_i)$$

This construction, illustrated in figure 2.2, is forward secure because the seeds cannot be inverted. If an attacker gains access to the current state, he cannot generate previous seeds.

Accordingly, the treehash algorithm in section 2.3.2 is modified to use a current seed, which is copied during its initialization and updated in each step. The current leaf number is not needed anymore.

The authentication path algorithm in section 2.3.3 reinitializes its treehash instances with the starting index $s+1+3\cdot 2^h$. The corresponding one-time seeds lie ahead of the current seed and hence need to be precomputed. For this purpose, the structure NextSeed is added to the state. It contains one seed for each treehash height $0,\ldots,H-K-1$. They are initialized with $\text{NextSeed}_h = \text{Seed}_{3\cdot 2^h}$ during key generation and updated once at the beginning of each authentication path computation. Consequently, when Treehash_h is reinitialized in step s, NextSeed_h matches the correct starting seed $\text{Seed}_{s+1+3\cdot 2^h}$.

In order to generate the current leaf in case $\tau = 0$, the authentication path algorithm additionally needs the current one-time seed, which can be passed as parameter from the signature generation function.

2.5 Tree Chaining

One of the main drawbacks of the Merkle scheme is that it allows only a limited number of signatures, which has to be defined in advance, and during initialization every Winternitz public key has to be computed. That means generating a tree with $H = 20$ providing $2^{20} = 1048576$ signatures and a public key creation time of 10 milliseconds would take nearly three hours.

This disadvantage can be countered by *Tree Chaining* (CMSS) proposed in [BGD+06]. The idea is to take T levels $1, \ldots, T$ of Merkle trees, using the lowest one to sign the actual data. All upper levels sign the root of the lower tree.

Given this chain of signatures, a verifier can successively reconstruct all roots up to the top level. Therefore the root of the highest level is placed in the public key.

Tree chaining allows fast initialization times with large-scale signature numbers, because only the first tree on each level has to be generated. This means for T levels with heights $H_1, \ldots, H_T$ only $2^{H_1} + \cdots + 2^{H_T}$ one-time public keys have to be processed during key generation instead of $2^{H_1 + \cdots + H_T}$ with a single tree. These heights can be different for each level just like the Winternitz parameters. This means the public key has to include T parameter pairs $(H, w)_i$. The Winternitz input x and bit masks, however, can be shared by all levels. As they might each need a different number of masks, their size is defined as the maximum required by all levels and each one uses just the part it needs.

2.5.1 Pending Tree

Each level except for T needs to replace the currently active tree at some time, when all its signatures are used. Therefore they have a pending tree that generates the next tree on that level distributed over all its signatures. With each signature of the active tree, one step is done on the pending tree, so that its computation is completed at the same time the active tree is depleted.

This level is refreshed by swapping active and pending tree. Then the signature on this level is updated by signing the root of the new active tree. Finally, the pending tree is reinitialized with a new starting seed to compute once more the next tree.

2.5.2 Signature Size

One of the drawbacks of tree chaining is the increased signature size, as a CMSS signature consists of T Merkle signatures. However, this is not as bad as it sounds, because the authentication paths have in fact together the same size as with a single tree of that height.

Furthermore, the upper signatures sign tree roots, which in practice have only half the size of messages, because they do not need collision resistance (see section 2.1.4).

The numbers s_i of the individual Merkle signatures $i = 1, \ldots, T$ can be combined to one overall signature number s. Starting with $t_T = s_T$ compute

$$t_i \;=\; t_{i+1} \cdot 2^{H_i} \;+\; s_i$$

for $i = T - 1, \ldots, 1$ and finally set $s = t_1$.

In the same way s can be decomposed into s_i by setting $j_1 = s$ and

$$\begin{aligned} s_i &= j_i \bmod 2^{H_i} \\ j_{i+1} &= j_i \;>>\; H_i \end{aligned}$$

for $i = 1, \ldots, T - 1$ and finally $s_T = j_T$.

2.5.3 Distributed Root Signing

Another problem with tree chaining is the more unbalanced signature generation time. In the worst case $T-1$ levels have to be refreshed, which means generating additional $T-1$ Merkle signatures. This includes computing new authentication paths, updating treehash instances and forwarding pending trees.

Therefore the authors in [BDK+07] present an extension called *Generalized Merkle Signature Scheme* (GMSS), which can be used to reduce this worst case signature time. The idea is to distribute the generation of the next authentication paths and root signatures on each level evenly between all signature steps. For this purpose, an additional tree is required between active and pending tree, increasing the size of the state by approximately one third. While the active tree is used and the pending tree is built, this new tree, called next tree, is already complete and its root is signed.

However, generating the next authentication path requires at most a single leaf node computation, without taking treehash updates into account, and signing the new root needs only half the operations of a one-time public key on average. This is only a fraction of the effort for treehash updates, which require a leaf node plus additional tree hashing each. Therefore distributing the root signing makes sense only with many levels, when at some step $T-1$ levels have to be refreshed.

My implementation is designed for a low number of levels, in order to keep signature sizes short. Therefore it does not use distributed signature generation. Instead, a different approach is used to distribute the treehash updates of all levels as described in the next section. This is more efficient since it requires no additional tree.

2.5.4 Distributed Updates

The uneven signature generation can be softened by distributing the updates for treehash instances and pending trees in upper levels over all signature steps. The update budget a tree receives for its treehash instances with each signature (see section 2.3.3) is just a maximum value. In fact not all of them are used.

The total number of updates over all steps is

$$2^H \cdot \frac{(H-K)}{2} = (H-K) \cdot 2^{H-1}. \tag{2.2}$$

The treehash instance on height i needs 2^i updates for a node, but has to compute only $2^{H-i-1}-2$ right nodes, because the first and second are saved during initialization.

There are instances on heights 0 to $H-K-1$ requiring therefore

$$\begin{aligned} & \sum_{i=0}^{H-K-1} ((2^{H-i-1}-2) \cdot 2^i) \\ = & \sum_{i=0}^{H-K-1} (2^{H-1} - 2^{i+1}) \\ = & \sum_{i=0}^{H-K-1} 2^{H-1} - \sum_{i=0}^{H-K-1} 2^{i+1} \\ = & (H-K) \cdot 2^{H-1} - \sum_{i=0}^{H-K-1} 2^{i+1} \end{aligned} \tag{2.3}$$

updates altogether. The leading part equals the number of available updates in equation 2.2.

Algorithm 2.5 Distribution of updates between levels of chained Merkle trees

```
updates ← (H−K)/2 + 1
level ← 1
while updates > 0 and level ≤ T do
    updates ← ActiveTree_level.Update(updates)
    while updates > 0 and ActiveTree_level.CurrentLeaf > PendingTree_level.CurrentLeaf do
        PendingTree_level.Step()
        updates ← updates − 1
    end while
    level ← level + 1
end while
```

Consequently, subtracting equation 2.3 from 2.2 gives an overall number of

$$\begin{aligned} & \sum_{i=0}^{H-K-1} 2^{i+1} \\ = & \sum_{i=1}^{H-K} 2^{i} \\ = & \sum_{i=0}^{H-K} 2^{i} - 1 \\ = & 2^{H-K+1} - 2 \end{aligned}$$

remaining updates.

The idea is to generate the next authentication path on upper levels without updating treehash instances and instead distribute the unused updates of the lowest level upwards in each step.

This can be combined with the pending tree steps, meaning we first do updates on the treehash instances and then advance the pending tree as long as it is behind of the active tree. The overall budget is therefore incremented by one, so that we can always do one step on the lowest pending tree, even if we used the maximal number of treehash updates before. Algorithm 2.5 implements this concept in pseudo-code.

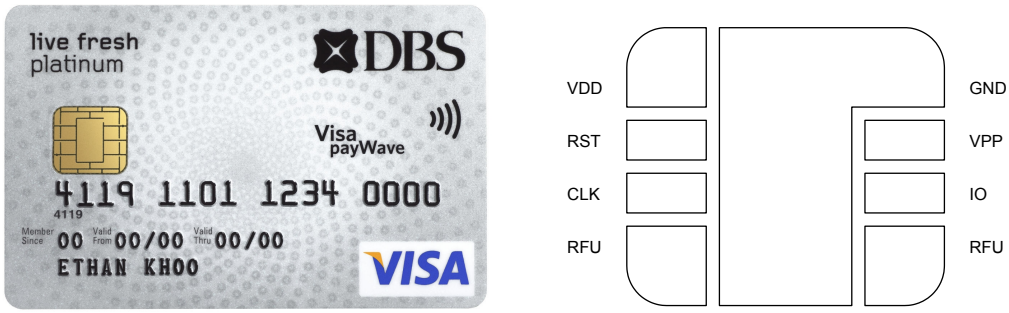
This works as long as $H - K$ is not too unbalanced between the levels. A simple requirement would be that the number of remaining updates in every level is at least as large as the number needed in the one above, meaning

$$2^{H_i - K_i + 1} - 2 \geq \frac{H_{i+1} - K_{i+1}}{2} + 1$$

for $i = 1, \ldots, T - 1$. This is, however, very conservative.

3 Smart Cards

The following chapter is about smart cards in general and the Infineon SLE78 in particular. Section 3.2 describes the protocols and parameters for communication between a smart card and an application on the host side. The architecture and features of the SLE78 family are detailed in section 3.3.



(a) Smart Card (b) Contact Pad

Figure 3.1: On the left is a typical smart card in form of an EMV credit card. On the right is a schematic of its standardized contact pad. The chip is powered via the supply voltage VDD and ground. VPP was used for EEPROM programming, which needs much higher voltages than the rest of the chip. Nowadays however, those voltages can be generated on the chip making this signal obsolete. CLK provides an external clock, RST stands for the reset signal and IO is used for communication, see section 3.2. The lower contacts are reserved for future use (RFU).

3.1 Basics

Smart Cards - also called Chip Cards or Integrated Ciruit Cards - are micro chips embedded in a plastic cover. They were introduced as replacement for magnetic stripe cards, which were very restricted in the amount of data that could be stored. Furthermore, the information on a magnetic stripe is unsecure, because it can be read, modified and copied to another card with little effort by anyone who has access to the card.

Smart cards in contrast are able to store, transmit and process data. They can be divided by functionality

- Memory cards have a fixed and simple logic in order to provide data storage, which is protected against unauthorized access
- Microprocessor cards feature a processor in order to execute programs, allowing them to provide user-defined applications

and by their interface

- Contact-based cards need to by physically connected via their contact pad
- Contact-less cards use a radio frequency (RF) field to both power the card and transmit data without the need for physical contact
- Hybrid cards provide both interfaces

Figure 3.1 shows an example of a smart card and its contact pad.

The first smart cards were memory cards used for example as prepaid telephone cards. With the introduction of microprocessor cards, their applications expanded to general authentication and identification. They are used in banking as

EMV (Europay, MasterCard and VISA) credit cards and in mobile phones as subscriber identity modules (SIM), just to name two.

There are two general forms of smart card usage: First for specific applications within and sometimes even across organizations, for example as identity card, passport or electronic purse. Second as general tokens for cryptographic operations like a Hardware Security Module (HSM) for a wide range of different applications using standardized interfaces like PKCS#11.

3.2 Communication

Communication between an application and a smart card is handled on the application level in form of *Application Protocol Data Units* (APDUs). APDUs are mapped onto *Transport Protocol Data Units* by a transport protocol in order to be sent over the serial interface. The response is then mapped back.

The structure of APDUs and the transport protocols are defined in the ISO standards 7816-3 [ISOa] and 7816-4 [ISOb]. These documents define two protocols:

- T=0 character-level transmission protocol
- T=1 block-level transmission protocol

For the host application, these protocols are generally handled by the smart card terminal, also called interface device. On the smart card they have to be implemented by the smart card operating system.

3.2.1 APDU Layout

There are two types of APDUs. Command APDUs are used to transfer a command message from the application to the smart card, while response APDUs carry the answer of the card back.

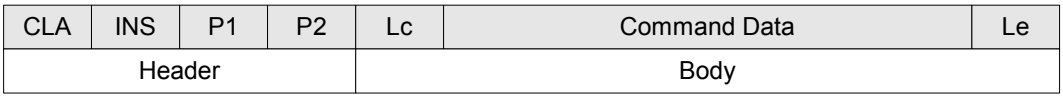


Figure 3.2: Structure of a command APDU, sent from the host to the card. The header determining the type of command is mandatory while the body is optional. The lengths of command and response data are given in bytes.

A command APDU consists of a header and an optional body as shown in figure 3.2.

The header encodes the type of the command with four bytes, the instruction class **CLA**, instruction code **INS** and two command parameters **P1** and **P2**.

The instruction code may not be 6x or 9x. In the original specification, they were further restricted to even values, because odd values were used by the card to signal the request for the programming voltage.

The body holds the **Command Data**. The length of this data field is encoded in one byte **Lc**. Finally, the length of the expected response data is given as byte **Le**.

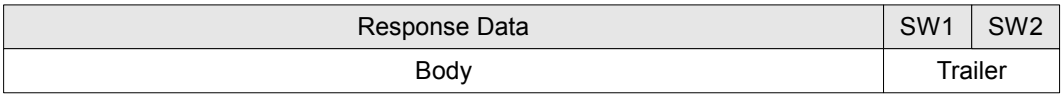


Figure 3.3: Structure of a response APDU, sent from the card to the host. The trailer containing the status code is mandatory while the body is optional.

The card answers with a response APDU as in figure 3.3. While the body containing the **Reponse Data** is optional, the two trailer bytes **SW1** and **SW2** are always returned.

The SW bytes are used to indicate errors with values 6xxx and 9000. Table 3.1 shows a classification of error codes. More detailed error descriptions are possible with different values for SW2. The meaning of this byte is however mostly application specific.

61xx and 9000	62xx	63xx	65xx	64xx	67xx to 6Fxx
Command executed			Command aborted		
Normal	Warning		Execution error		Parameter error
	NVM not modified	NVM modified	NVM not modified		

Table 3.1: The concept of return codes as specified in the ISO documents. A successful command is confirmed with 9000. 61xx is reserved for T=0.

The layouts of command and response APDUs have optional parts. If a command contains no additional command data, this data and its length field are left out. In the same way the response data and expected length fields are present only if there is data supposed to be transferred back from the smart card (except from the status code).

This leaves four different cases for the structure of an APDU:

- Case 1: No command data is sent and no response data expected. In this case, the command APDU only consists of the header, the response APDU only of the trailer.
- Case 2: No command data, but response data is expected. In this case, the command APDU consists of the header and Le field, the response APDU of the data field and trailer.
- Case 3: There is command data, but no response data. In this case, the command APDU consists of the header, Lc and data fields, the response APDU only of the trailer.
- Case 4: There is command data and response data. In this case, the command APDU consists of the header, Lc, data and Le fields, the response APDU of the data field and trailer.

3.2.2 T=0 Protocol

With the character-level transmission protocol T=0, communication is always initiated by the terminal and a data block can be transmitted only in one direction.

A command TPDU has a five byte header, where the first four bytes are taken directly from the APDU as CLA, INS, P1 and P2. The fifth byte is called P3 and represents the size of the data block that is supposed to be transferred subsequently. The direction of this transfer depends on the instruction and is supposed to be known to both card and terminal in advance.

A response TPDU looks exactly like a response APDU and therefore is always returned unchanged.

The four APDU cases are handled as follows:

- Case 1: The command is sent with P3 = 0 and no data block is transmitted.
- Case 2: The command is sent with P3 = Le, where Le is valued from 1 to 256 and 256 is encoded as 0. The data block is transmitted in the response TPDU.
- Case 3: The command is sent with P3 = Lc, where Lc is valued from 1 to 255. The data block is transmitted in the command TPDU directly after its header.
- Case 4: This involves a bidirectional data transfer and cannot be done with a single TPDU. Therefore the command is sent first with P3 = Lc, as in case 3. The card then answers with SW = 61xx instead of 9000, where xx denotes the number of bytes available and 256 is again encoded as 0. The opposite side is then supposed to issue a *GET RESPONSE* APDU with the special instruction code C0 requesting to receive that data like in case 2.

Accordingly, the maximal transmission size with T=0 is 255 byte for sending and 256 for receiving data.

After sending the five header bytes, the terminal waits for a procedure byte from the card. There are three types of procedure bytes:

- **ACK** = INS: The card acknowledges a command by echoing its instruction code. Depending on the case, the terminal then starts transmitting or receiving the data block.
- **NULL** = 60: When receiving a NULL byte, the terminal resets the wait working time and waits for the next procedure byte. The wait working time is the maximal duration the terminal is waiting for an answer.
- **SW1** = 6x or 9x (except 60): The card can show its disapproval with a command by sending back an error code before exchanging the data block. In that case, the terminal waits for the SW2 byte to complete the communication. This can be distinguished from an acknowledgment because instruction codes may not be 6x or 9x.

3.2.3 Protocol Type Selection

The ISO standards define a set of parameters that affect the communication between a card and the terminal. They specify, for example, the programming voltage and the type of protocol used. There are also protocol specific parameters including things like data rate and block waiting time.

Directly after power up, the smart card sends out a sequence of bytes called *Answer To Reset* (ATR). They indicate the initial parameters it assumes for the further communication and a set of parameter combinations it supports.

If the card offers more protocol types or parameter values than the initial ones, the interface device can start the *Protocol Type Selection* (PTS) process. First the terminal sends a PTS request selecting one of the supported protocol and parameter combinations, which is acknowledged by the card with a PTS confirm. Finally, both change their communication parameters accordingly.

The PTS handshake has to be initiated right after the ATR and before any other command. Otherwise communication continues with the initial parameters, as they cannot be changed in the middle of a session.

3.3 Infineon SLE78

The SLE78 is a high-end security smart card developed by Infineon Technologies with Common Criteria EAL5+ certification. Figure 3.4 shows its architecture in a block diagram. All following information including the diagram are taken from the Programmer's [Inf11] and the Hardware Reference Manual [Inf10] for the SLx70 family.

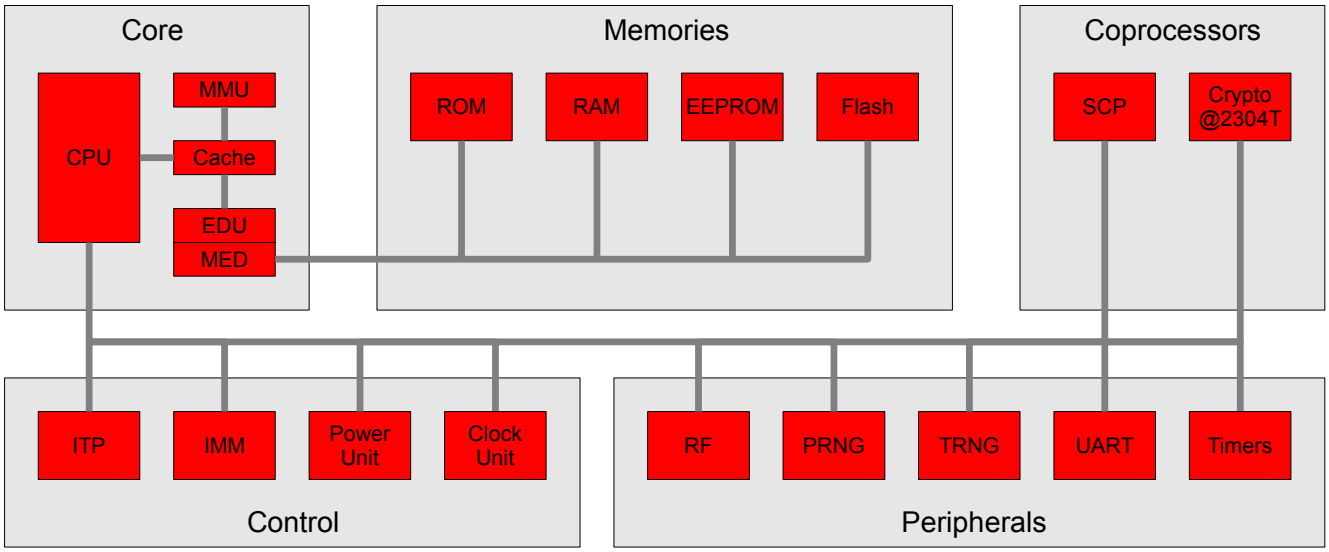


Figure 3.4: Block diagram of the SLE78 security controller. It contains a 16-bit CPU with cache and a memory management unit for virtual addressing. The memories are protected by encryption and error detection units. Communication includes both contact-based and contact-less interfaces. For cryptographic purposes the SLE78 features coprocessors and random number generators.

The core consists of a 16-bit CPU running at 33 MHZ, a single-cycle access cache and a memory management unit (MMU) supporting up to 24-bit virtual addressing. Memory access is protected against errors and attackers by a Memory Encryption/Decryption (MED) and an Error Detection Unit (EDU). The instruction set includes 16-bit integer instructions with some 32-bit extensions.

There are three types of memories: Read Only Memory (ROM) is used for the firmware and sometimes operating systems, Random Access Memory (RAM) for temporary data storage and Non-Volatile Memory (NVM) in the form of EEPROM respectively Flash for persistent data and code. The sizes of those memories depend on the derivative.

The control part consists of an Interrupt and Peripheral Event Controller (ITP) together with interface (IMM), power and clock management.

For fast cryptographic computations, the SLE78 features coprocessors for both symmetric (SCP) and asymmetric (Crypto@2304T) cryptography.

Futhermore, a number of peripherals is supported, starting from security instruments (such as sensors) to Pseudo and True Random Number Generators (PRNG and TRNG), Timers and contact-based (UART), as well as contact-less (RF) communication. The architecture allows even more peripherals and interfaces not shown here like I2C and USB. What peripherals are included depends on the derivative.

I worked with a SLE78CFLX4000PM offering 8 KB RAM, 404 KB NVM, a True Random Number Generator and contact-less interface.

3.3.1 Symmetric Crypto Processor

The Symmetric Crypto Processor (SCP) offers hardware acceleration for the standardized encryption algorithms DES, Triple-DES and AES. As we will see later, the primitives in section 2.1 can be implemented with a block cipher. Therefore the SCP is the most essential part regarding performance.

AES supports key sizes of 128, 192 and 256 bit, but in contrast to the original Rijndael specification unfortunately only a block length of 128 bit. This limits its use to an output length of 128 bit, at least with single-block-length constructions.

3.3.2 True Random Number Generator

As said in sections 2.3.1 and 2.4.1, a complete Merkle tree is generated from one starting seed. As soon as one gets this seed, one can reconstruct the complete key, including every single Winternitz secret key.

Therefore it is important that this starting seed is chosen truly random. Its entropy determines an upper bound for the security level of the scheme. For example, if this was only 64 bit, an attacker could check all 2^{64} possible seeds to find the one that was used to generate the key.

The True Random Number Generator included in the SLE78 should be applicable for this function, as it meets the quality criteria defined in AIS-31 [SK].

3.3.3 Non-Volatile Memory

Non-Volatile Memory (NVM) is used to store persistent data that must be available in the next session, even when the card is unpowered. It is organized in sectors which consist of 32 logical pages plus 1 spare page. The size of one page is 256 byte with the given derivate.

While reading from NVM is rather uncomplicated, writing requires a special firmware function and a page is always written as one. This means even when changing only one byte in a page, that (logical) page has to be completely reprogrammed. Physically, the spare page is programmed with the updated content. The original page is erased thereafter and becomes the new spare page. The mapping between physical and logical pages is handled by the firmware.

Due to the physical nature of the NVM, a physical page can only be programmed a limited number of times, called cycle count. This figure is about 500000 with the given Flash technology. After that, the page will wear out and cannot be programmed reliably anymore.

Furthermore, programming a page disturbs all other pages in that sector a bit. Therefore every occupied page has to be refreshed after about 1000 writes to neighbouring pages, in order to prevent it from loosing its content. This is called disturbance handling and is done automatically by the firmware.

Through spare page rotation and disturbance handling, the programming operations (cycle stress) are shared among all 33 physical pages of a sector. Therefore the overall cycle count of a sector is approximately 16.5 million.

4 Implementation

In this chapter, I present my implementation of the Extended Merkle Signature Scheme. Through the following pages, MSS and Merkle will always refer to the extended instead of the classical version of the scheme.

The Keil MicroVision development environment and compiler tool chain provided by Infineon include a simulator, which can be used to debug a smart card program. It also has an interface to the outside, so that a host application can communicate with the simulated card. This way, the correct operation of the signature scheme can be verified within the development environment.

The smart card software is written completely in C. This leaves some room for improvements by replacing certain parts with hand-optimized assembler code. The most critical part performance-wise is however the AES implementation, which is done by the coprocessor and can hardly be optimized any more.

Implementing the scheme on the smart card presents some challenges:

- **Resources**: Smart cards have only very limited resources like RAM, while Winternitz signatures and key pairs can become quite large.
- **Flexibility**: The Merkle Signature Scheme has four key parameters: The height H, retain height K, Winternitz parameter w and message size m. As different applications might need different parameters, the implementation should be dynamic. That means the parameters can be chosen freely and the memory requirements are adapted accordingly. This should, however, not waste too many resources.
- **Cycle Stress**: The private key in form of the state has to be stored in NVM. Unfortunately, the algorithm has to update its state in every step and at multiple locations. As the card might get unpowered after each signature, those modifications always have to be written to NVM. This costs not only time but also valuable cycles and requires an efficient NVM layout in order to reduce the cycle stress.
- **Initialization Time**: During key generation, the whole Merkle tree has to be computed. The costs thereby increase exponentially with the height of the tree. For a large number of signatures, tree chaining is needed to keep the initialization time reasonable.

The following chapter shows how these challenges are solved.

Section 4.1 starts with the ASN.1 structure used to encode the data returned by the card. It is chosen in a way that permits encoding the data partly. That means the first part can be sent with its ASN.1 tag and length information before the second is computed. Therefore we never have to hold the signature in RAM completely.

The next section 4.2 is about memory management for both RAM and NVM.

It is followed by section 4.3, which deals with the communication between application and smart card and the interface provided by the smart card operating system.

Section 4.4 describes the architecture and implementation of the smart card software starting with the primitives. It continues with the implementation of the Winternitz one-time signature scheme. The next part is the treehash algorithm used for initialization and for updating the treehash instances. These modules are used to construct the single tree Merkle scheme. Finally tree chaining is built on top of that.

Section 4.5 gives a short overview of the host implementation on the opposite side.

4.1 ASN.1 Encoding

In order to exchange signatures and public keys between chip card and host (or any other system for that matter), they are encoded in ASN.1, as it is common in cryptography. From the encoding perspective, a single tree is just a special case of tree chaining. Therefore the definitions below apply to both interfaces. The normal MSS is just regarded as tree chaining with only one level.

4.1.1 Signature

A signature consists of the signature number and a chain of signature levels, encoded in highest-level-first order. A signature level is again a pair of authentication path and Winternitz signature for one level. In ASN.1 this looks as follows:

```
mss_sig_level ::= SEQUENCE {
    auth_path  OCTET STRING,
    wots_sig   OCTET STRING
}

mss_sig ::= SEQUENCE {
    s          INTEGER,
    sig_chain  SEQUENCE OF mss_sig_level
}
```

The authentication path and Winternitz signature could be described in more detail as a sequence of nodes, but that would blow up the encoding size overhead, as each node would need a header with tag and length. That is why they are defined as simple octet strings.

When decoding a signature, its number of tree levels can be obtained as the number of elements in the signature level sequence. The definition as *SEQUENCE OF* additionally demands, that at least one level must be present in a valid signature encoding.

4.1.2 Public Key

The public key has to contain all information needed to verify a signature.

This includes the message size m and root of the top level tree. In contrast to the definition in chapter 2 the message size is given in byte, not in bit. Using an arbitrary number of bits would just make things unnecessarily complicated.

Height and Winternitz parameter are defined for each level in an own sequence just like the signature level above. Its number of elements again determines the number of levels, at least one must be included for the encoding to be valid and the highest level comes first.

The bit masks and Winternitz input x are the same for all levels.

Furthermore, the type of PRF and hash function used for Winternitz and Merkle needs to be included. This is done with an ASN.1 object identifier which represents a specific pair of functions.

The public key definition therefore finally looks like:

```
mss_pk_level ::= SEQUENCE {
    height     INTEGER,
    w          INTEGER
}

mss_pk ::= SEQUENCE {
    level      SEQUENCE OF mss_pk_level,
    algorithm  OBJECT IDENTIFIER,
    m          INTEGER,
    root       OCTET STRING,
    x          OCTET STRING,
    xor_mask   OCTET STRING
}
```

4.1.3 Algorithm Identifier

The algorithm identifiers are placed under the object space *1.3.6.1.4.1.8301.3.1*.

On the card, the algorithm will always be AES128 for both PRF and hash function, in order to use the hardware acceleration. The corresponding ASN.1 object value can therefore be hard coded into the program.

The host implementation is more flexible and constructs its function pointer structure from this object identifier.

4.2 Memory Management

Of the 8KB RAM the Infineon smart card offers, 320 byte are reserved by the firmware. Another 256 byte are needed for the program stack and around 64 byte for things like global variables. This leaves a maximum of 7552 byte as heap space.

The SDK provides a heap management implementation in the standard C library via the functions malloc and free. It needs 6 byte overhead per allocated and free block, 2 byte for the size and 4 for a pointer to the next free block. As my application frees its memory generally in the same order it was allocated, there is only one free block and no fragmentation. This means the overhead for organizing the list linking the blocks should be small.

NVM is accessed by mapping it linearly into the address space of the application. Managing this with some kind of file system is usually the job of a smart card operating system. To keep it simple, my application allocates a large enough part of NVM space and uses a simple and yet efficient layout in order to save its data. The design criterion is to minimize cycle stress without wasting too much space.

The key initialization method expects a page-aligned NVM address, where it places a control structure followed by the remaining structures. Finally, it returns an address marking the end of its allocation. This control structure contains pointers to all subsequent structures, therefore the other functions only need the address of the control structure.

The NVM layout is described in section 4.4.5 for a single tree and in 4.4.6 for tree chaining.

4.3 Communication

The fundamentals of smart card communication are explained in section 3.2. My implementation uses the T=0 protocol because it is easy to implement.

The smart card operating system basically consists of a loop receiving, processing and answering commands from the host.

As the implementation focuses on the signature scheme, Protocol Type Selection (section 3.2.3) is not included. The ATR is specific and as simple as possible, so that it works with any reader. With these settings, the baud rate determining the communication speed between card and terminal is very low. However, as the SLE78 supports more than 1000 kilobaud, communication times should generally not be a bottleneck.

4.3.1 Interface

The application interface basically consists of three functions:

- Key generation takes the necessary parameters to generate and initialize a new signature scheme instance, it does not return any data
- Public key extraction encodes and returns the public key of the current instance
- Signature generation expects the message to be signed and returns its signature

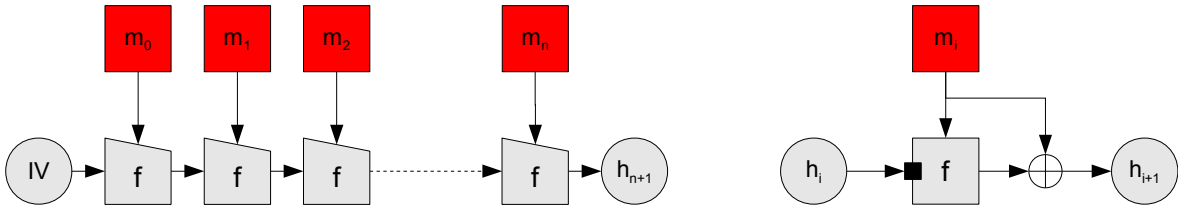
Each of them has an own instruction code.

4.3.2 Transmission Protocol

Signatures and public keys returned by the card will become much larger than the 256 bytes allowed by the T=0 protocol.

This requires an extension to the protocol. When handling a case 4 APDU, the smart card has to signal that it wants to send data with the return code 61xx. This method is simply extended.

In order to transmit more than 256 bytes, the data is divided into parts smaller or equal to 256 bytes. After the first part, the card answers again with 61xx, where xx is the length of the next part to be transferred (with 256 encoded as 0). This step is repeated for all parts.



(a) Merkle-Damgard construction (b) Matyas-Meyer-Oseas

Figure 4.1: This figure shows the construction of a hash function from a compression function as proposed by Merkle and Damgard. The message is split in n blocks and each block is combined with the output h_i of the previous iteration, starting from an Initialization Vector IV $= h_0$. The hash value corresponds to the final iteration's output h_{n+1}. A block cipher in Matyas-Meyer-Oseas conjunction can be used as compression function.

Accordingly, the opposite side is expected to send *GET REPONSE* TPDUs requesting the indicated length until the return code is different from 61xx. At the end, all parts are concatenated and the last SW is returned.

This procedure is modified by sending the overall length of the data transfer encoded in two bytes as first part. This allows the host application to check the size or allocate enough memory dynamically before continuing with the remaining transmission.

4.3.3 Keep Alive

As explained in section 3.2.2, the T=0 protocol has a timeout called work waiting time. If the card does not answer within this time span, the interface device assumes a communications error and resets the card.

The exact value of the work waiting time depends on the communication parameters. Usually this will be something below one second. However, at least key generation on the card can take some time, which is certainly longer than a second.

In order to avoid this timeout, the card has to request an extension of the work waiting time by sending a NULL byte before it runs out. This can be implemented as a form of keep alive signal with one of the timers provided by the SLE78.

Each timer has a 16-bit counter value and can be attached to the external clock, the 33 MHZ system clock or an internal 1 MHZ clock. The prescale factor defines the number of clock cycles after which the counter is increased by one. Using the 1 MHZ clock and a prescale factor of 64, the timer does about 781 steps in 50 milliseconds.

The timer can be configured to issue an interrupt request whenever its value overflows, triggering an interrupt service routine, which transmits the wait time extension byte and resets the timer.

4.4 Architecture

This section describes the implementation of the Extended Merkle Signature Scheme and its components defined in chapter 2.

There are two different versions, one for a single tree and one for tree chaining. This is mostly due to optimization purposes. The main part including the primitive functions, Winternitz, treehash instances and other basic code is the same for both.

4.4.1 Primitives

The primitive functions are defined in section 2.1 and implemented with AES128 using the symmetric coprocessor. Therefore the length of Winternitz data blocks, seeds and tree nodes is always the same with 16 byte.

The coprocessor is accessed via 16-bit data transfers. This means loading a key, input or output needs eight reads respectively writes. As AES is the kernel function, using a loop for those transfers is undesirable. Therefore key loading is done in eight subsequent instructions. Encryption is done in the same manner: Write the input, wait for the coprocessor to finish and read the output back.

Algorithm 4.1 Saving nodes during initialization

```
function TREE.SAVENODE(node, h)
    index ← ((CurrentLeaf + 1) >> h) − 1          ▷ The index of this node on its height
    if index = 1 then
        Auth_h ← node
    end if
    if index = 3 and h < H − K then
        TreehashNode_h ← node
    end if
    if index ≥ 3 and index is odd and h ≥ H − K then
        Retain_h.push(node)
    end if
end function
```

The PRF is implemented straightforward by encrpyting the input x with AES using key k. Its output corresponds to the ciphertext. The PRG and FSPRG need to load the seed as key only once and then do as many encryptions as needed. As 65536 iterations are far more than necessary, the PRG counter is implemented with one word only.

The hash function is built with the Merkle-Damgard construction using a compression function as illustrated in figure 4.1(a). Compression functions can in turn be implemented with block ciphers. One of the most popular designs for this is Meyer-Matyas-Oseas shown in figure 4.1(b). It encrypts the chain input with the message block as key. The ciphertext is then XORed with the original message block and forwarded to the next iteration.

In general cases with variable-length inputs, a MD-compliant padding has to be appended as a last step before the final hash value in order to satisfy the security proof for the Merkle-Damgard construction (see section 8.5 in [GB08]). Merkle originally proposed a length padding, where the number of data blocks in the message is encoded as additional data block and appended as finalization step. However, due to the fixed input length of the hash function, a padding is unnecessary in this implementation.

Therefore the hash function looks like

$$\text{hash}(l \parallel r) = \text{AES}_{\text{AES}_{\text{IV}}(l) \oplus l}(r) \oplus r$$

where l and r correspond to the left and right child node and IV is the initialization vector. At the moment, this is set to all zero.

This means we need only two evaluations of the compression function and hence only two AES operations in order to compute a parent node in the hash tree.

4.4.2 Winternitz OTS

A Winternitz instance is defined by its parameter structure, which contains the message size m, Winternitz parameter w and input x as well as the lengths l, l_1 and l_2.

In order to compute those lengths, logarithm evaluations with floating point accuracy are necessary (see section 2.2), which is sligthly problematic on a smart card without FPU. It is possible to use floats and the log function of the standard math library, however this expanded the code size by about 2 KB. Implementing floating point operations on integer ALUs is of course not efficient. But as this initialization is done only once during key generation, the performance impact should be insignificant.

Calculating the exponents b_i requires subsequently dividing the message, interpreted as one integer, by w. One complete division is done with m 16-bit integer divisions, as those are the largest available in hardware. Provided that the message is saved in bytes $M_1, \ldots, M_m$ in big-endian byte order, this can be implemented iteratively as

$$\begin{aligned} M'_i &= ((r_{i-1} << 8) \mid M_i) \;/\; w \\ r_i &= M_i - w \cdot M'_i \end{aligned}$$

for $i = 1, \ldots, m$ where r_i are bytes, $r_0 = 0$ and $\mid$ denotes the bitwise OR operation. Finally the quotient is in M'_i and the remainder in r_m.

Algorithm 4.2 One step for key generation

```
function TREE.STEP
    h ← NextSeedHeight
    if CurrentLeaf = 3 · 2^h and h < H − K then
        NextSeed_h ← CurrentSeed
        NextSeedHeight ← NextSeedHeight + 1
    end if

    (otsSeed, CurrentSeed) ← fsprg(CurrentSeed)
    node_1 ← GenLeafNode(GenPK(otsSeed))
    height ← 0
    SaveNode(node_1, height)

    while height < ctz(CurrentLeaf + 1) do        ▷ Number of trailing zeros in the binary representation
        node_2 ← Stack.pop()
        node_1 ← GenParentNode(node_1, node_2)
        height ← height + 1
        SaveNode(node_1, height)
    end while

    Stack.push(node_1)
    CurrentLeaf ← CurrentLeaf + 1
end function
```

As the maximal value of the checksum is $C \leq l_1(w-1)$, a 16-bit variable here should be sufficient even for large messages. This allows to divide the checksum with one hardware division.

4.4.3 Initialization

Generating a tree means computing its root and filling the structures Auth, TreehashNode and Retain with the required nodes. This is done similar to the update function of the treehash algorithm in 2^H steps, except that every time we pass through a node, we check if it has to be saved, see algorithm 4.1.

Please note that Retain_h is here not a stack but a FIFO queue (first in, first out), because the nodes are encountered during tree generation in the same order as they are needed in the authentication path algorithm.

In practice, Retain can be implemented without having to manage queues nor stacks by simply saving all nodes in one array with $2^K - K - 1$ elements. This is the number of retain nodes, see equation 2.1. The position of a node in this array is determined by its height and index as

$$2^{\overline{h}} - \overline{h} - 1 \; + \; (\text{index} \; >> 1) \; - 1$$

where $\overline{h} = H - h - 1$ is the number of retain heights previous to h. This is the reverse height minus one, because the first retain height is $H - 1$.

Hence $2^{\overline{h}} - \overline{h} - 1$ is the offset for this height, that is the sum of all retain nodes from previous heights. The remaining part is the number of this node on its height. The index is divided by two, because only right nodes are considered, and one is subtracted, because the first right node is not included.

One step during tree generation corresponds to computing the next leaf node as shown in algorithm 4.2. Meanwhile, SaveNode is used to fill the initial authentication path, treehash instances and retain nodes. Furthermore, the current seed is saved into NextSeed_h, if the current leaf is $3 \cdot 2^h$.

The variables CurrentLeaf and NextSeedHeight should be initialized with 0, while CurrentSeed should to begin with a random starting seed.

Having to remember the height of all nodes on the stack can be avoided, because the number of nodes which are combined in each step corresponds to the number of trailing zeros *ctz* in the binary representation of CurrentLeaf + 1.

Algorithm 4.3 Modified treehash functions with shared stack and forward secure seed computation

```
function TREEHASH_h.INITIALIZE
    TreehashSeed_h ← NextSeed_h
end function

function TREEHASH_h.UPDATE
    (otsSeed, TreehashSeed_h) ← fsprg(TreehashSeed_h)
    node_1 ← GenLeafNode(GenPK(otsSeed))          ▷ Compute public key directly from one-time seed
    TailHeight_h ← 0

    while TailHeight_h = height of top node on Stack and TailHeight_h + 1 < h do
        node_2 ← Stack.pop()
        node_1 ← GenParentNode(node_1, node_2)
        TailHeight_h ← TailHeight_h + 1
    end while

    if TailHeight_h + 1 < h then
        Stack.push(node_1)
    else
        if TreehashNode_h is used then
            node_2 ← TreehashNode_h
            node_1 ← GenParentNode(node_1, node_2)
            TailHeight_h ← TailHeight_h + 1
        end if
        TreehashNode_h ← node_1
    end if
end function
```

4.4.4 Treehash

The Treehash instances are used to compute right nodes for upcoming authentications paths. The algorithm is described in section 2.3.2.

All instances share a single stack while saving one node seperately. The latter nodes are stored in the structure TreehashNode. The current seed of each instance represents the structure TreehashSeed.

Additionally, a TailHeight is needed, which encodes the current state of the instance and the height of the lowest tail node. Both information can be packed in one byte, 3 bit encode the state and the remaining 5 bit the tail height.

The state determines if the instance is

- new, meaning it has been initialized with a new seed but not received any update yet, or
- running, meaning it has received at least one update, or
- finished, meaning it is not needed anymore.

The tail height of a instance is defined as

$$\text{Treehash}_h.\text{height} = \begin{cases} h & \text{if Treehash}_h \text{ is new} \\ \text{TailHeight}_h & \text{if Treehash}_h \text{ is running and TailHeight}_h < h \\ \infty & \text{if Treehash}_h \text{ is finished or TailHeight}_h = h \end{cases}$$

The value ∞ is used to exclude finished instances from the update process, see section 2.3.3.

Furthermore, the state indicates if the internal node is used. This information is needed in the modified update function shown in algorithm 4.3.

The state, tail height and stack heights could also be replaced by counting the leaf number per instance and using the number of trailing zeros like during initialization. This is, however, not worth the effort in this case, since the shared stack needs only $H-K-2$ heights and the state including tail height another $H-K$ byte. Saving the leaf numbers would require 16-bit words and therefore $2 \cdot (H-K)$ byte.

Algorithm 4.4 Signature generation for a single tree in principle

```
function TREE.SIGN(message)
    if CurrentLeaf = 2^H then
        return error
    end if

    Output Auth and CurrentLeaf

    Load control structure into RAM
    (otsSeed, CurrentSeed) ← fsprg(CurrentSeed)

    Load update section into RAM
    GenNextAuth(otsSeed)
    Update((H − K)/2)
    Write update section to NVM

    CurrentLeaf ← CurrentLeaf + 1
    Write control structure to NVM

    Sig_ots ← OTS.Sign(message, GenSK(otsSeed))
    Output Sig_ots
end function
```

4.4.5 Single Tree

During initialization, the control structure has to be allocated and initialized. This structure contains the current leaf number, current seed, Winternitz parameters and the tree root. It further has pointers to the structures Auth, Keep, Retain, NextSeed, the treehash data TreehashNode, TreehashSeed and their shared Stack as well as the Masks.

As the state of the treehash instances and the height information of the stack are only a few bytes, they are saved inside the control structure with a fixed size. This saves NVM space and cycles, because the page with the control structure is modified and written in every signature step anyway.

With a single tree key generation means obtaining a starting seed from the True Random Number Generator and performing 2^H steps with algorithm 4.2. Thereafter the root is saved and the key is ready.

The process of a signature generation is demonstrated in algorithm 4.4. In each step, the algorithm state is updated as described in section 2.3.3. Some of the information saved directly in the control structure is modified multiple times and at different points in time. Therefore it is copied into a RAM buffer at the beginning and written back to NVM once at the end.

The authentication path and leaf number always correspond to the current signature step. Consequently they can be encoded and sent over the communication channel first. This can even include the tag and length information of the subsequent Winternitz signature. That is the reason why authentication path and leaf number are placed at the beginning of the ASN.1 sequence defined in section 4.1.1.

The next step is to forward the current seed, which outputs the one-time seed corresponding to this leaf. Prior to this however, the state is updated by generating the next authentication path and distributing updates between the treehash instances. Then the control structure is written back to NVM.

Finally, the one-time seed is used to generate the Winternitz secret key, which in turn is used to sign the actual message. The Winternitz signature is then sent as second stage of the communication process, which also completes the signature step as a whole.

Sending the Winternitz signature after the state is written permanently to NVM ensures that an attacker cannot obtain a valid Winternitz signature, then interrupt the process before the current seed is updated (for example by unpowering the card) and eventually generate a new signature for a different message with the same one-time key in the next session.

Let us take another look at the update process in section 2.3.3: The authentication path algorithm 2.2 forwards the NextSeeds, saves a node in Keep every second time, reinitializes some TreehashSeeds and updates the leading part of the authentication path. Each updated treehash instance forwards its TreehashSeed and either saves a node inside TreehashNode or on the shared Stack. Often, a single instance receives more than one update in a step.

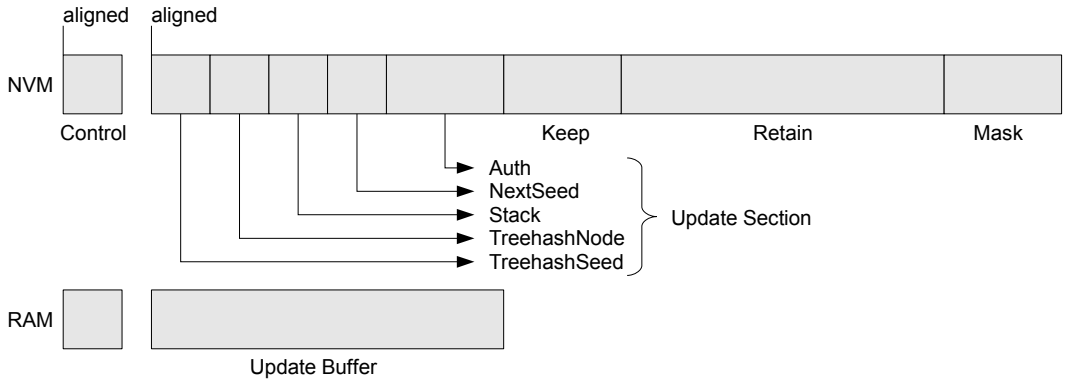


Figure 4.2: NVM data layout for a single tree. It starts with the control structure followed by the Update Section consisting of TreehashSeed, TreehashNode, Stack, NextSeed and Auth, aligned to the beginning of a page. Keep, Retain and the Masks are placed directly after that.

Writing every modification directly to NVM is not a good idea. Instead, the temporal and spatial locality of these write accesses should be utilized by caching them in a RAM buffer and executing them at once. For that reason, the structures that are updated together - TreehashSeed, TreehashNode, Stack, NextSeed and the authentication path - are placed next to each other in the NVM layout, as shown in figure 4.2.

Aligning the start to the beginning of a page ensures that the complete update section occupies a minimal number of pages. As all structures except control consist only of nodes and the page size is a multiple of the node size, they are subsequently all aligned to a node.

The authentication path is placed at the end, because it is modified only partly. The first node is replaced in each step, the second in every second, the third in every forth and so on. Hence when the authentication path overlaps two pages, the second one does not need to be programmed in every step. Depending on the size of the authentication path and the structures in front of it, this will probably save some additional write cycles.

Keep is not included, because it is modified only every second time and then just with a single node. The Retain and Mask structures both remain constant over all signatures.

4.4.6 Tree Chaining

The tree chaining version implements the concept explained in section 2.5. On the smart card it is restricted to two tree levels, allowing a smaller and more optimized implementation.

The control structure for tree chaining includes control structures for the upper and lower tree. Additionally, it contains the current state of the pending tree including the stack used for building. As said before, the Winternitz input x and the masks are shared between both levels.

During initialization, a tree is generated on both levels together with the corresponding signature of the lower root. All starting seeds are obtained on demand from the True Random Number Generator.

When the lower tree is finished, it has to be refreshed. This means copying the initial authentication path, TreehashNodes and NextSeeds from the pending tree. Furthermore, the upper signature is updated with the new lower root and the pending tree is reinitialized with a new starting seed.

In contrast to the original construction, the authentication path of the upper tree is always updated before the next signature, not after the current. This way, the upper authentication path always matches the current upper signature. Therefore it can be taken directly from the upper state and does not need to be saved additionally.

Signature generation is similar to the single tree. In the signature chain of the ASN.1 definition in section 4.1.1, higher signatures are placed first. Therefore the upper signature can be transmitted first together with the signature number and both authentication paths.

Then the next lower authentication path is computed and updates for treehash instances are distributed over both levels as in section 2.5.4. With two levels, the only pending tree is the one on the lower level, which always receives exactly one update per step. Therefore the update algorithm can be simplified.

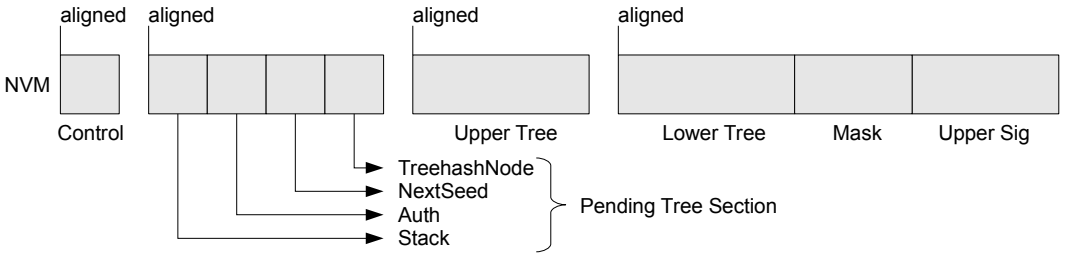


Figure 4.3: NVM data layout for two chained trees. The control structure is followed by the pending, upper and lower tree. Finally the mask shared by both and the upper Winternitz signature are placed at the end.

The final step is again computing and transmitting the Winternitz signature for the input message.

With tree chaining, the state consists of the upper, lower and pending tree as shown in figure 4.3.

The pending tree needs space for its stack, together with the initial authentication path, TreehashNode and NextSeed. The retain nodes can be written directly into the lower tree Retain structure, because at the time the pending tree gets to a retain node it has already been used by the active tree and is not needed anymore. As the pending tree stack is updated and hence written in every step, it should be aligned to a page so that it occupies a minimal number of pages. In practice, the control structure needs the complete first page anyway.

The upper and lower tree are placed behind the pending tree. They use the same layout as with a single tree in figure 4.2 except for the control structure, which is already included in the control structure, and the masks, which are shared by both levels. Both trees are aligned to the start of a page so that their update sections are aligned.

The shared masks are positioned at the end together with the upper Winternitz signature.

4.5 Host

The host side application implements the complete Merkle Signature Scheme including tree chaining with an arbitrary number of levels. The scheme itself uses an abstract interface for PRFs and hash functions, implemented in C with function pointers. That means you can easily add own function combinations to the list of algorithm identifiers in section 4.1.3.

Tree chaining is based on top of the single tree functions. They use the same data structures for signatures and public keys, which allows the use of one verification function for both. ASN.1 is handled by functions for encoding and decoding signatures and public keys using the helper functions of the OpenSSL library.

The communication interface to a card is written in C++. It uses an abstract class to implement the basic interface code, including the extended transmission protocol from section 4.3.2 using a virtual function to send APDUs.

This class is then used to derive a hardware version, which uses the PCSC interface in order to connect to a card reader and implement the APDU transmission function. In the same way, another class is derived, that handles the connection and communication with the simulator. This allows the use of the same application code interchangeably with a real and a simulated card.

The implementation of the signature scheme itself is platform independent. This means the host application can be ported easily to a linux operating system, provided it has a PCSC implementation.

5 Results

After successfully implementing the signature scheme using the simulator, I finally had the chance to run it on hardware. This chapter illustrates the resulting performance.

The following sections show the run times for key and signature generation on hardware. They also give the times resulting from the profiling function of the simulator, those are however not really accurate. Furthermore, the NVM space required for the state is evaluated together with the size of keys and signatures. Finally, the NVM cycle stress is estimated for one particular parameter combination.

This analysis is done separately for both the single tree and tree chaining variant of the implementation in section 5.1 and 5.2 respectively. Section 5.3 continues with the run times for signature verification on the card.

Finally, the scheme implemented in this thesis is examed briefly for its resistivity against side-channel attacks in section 5.4.

The presented run times are computation times, which means they do not include communication overhead. Therefore the transmission of the signature is omitted during measurements. With the simulator, the profiling can be done directly for a specific function. The run times on hardware are however measured from the host side. In order to get the remaining communication overhead, I used an empty command that does nothing more than transmitting the data to be signed to the card. This takes about 0.06s with a data size of 32 byte. This time is already subtracted from the signature generation and verification numbers presented here. For the key generation it is insignificant.

As described in section 4.1, public keys and signatures returned are encoded in ASN.1. Therefore their sizes are given in both raw and encoded form in order to determine the overhead of the ASN.1 encoding. The raw size of a public key includes the masks, one byte for each height, Winternitz parameter and message size plus 32 byte for the root and Winternitz input. The encoding adds 10 byte for the algorithm identifier and of course the overhead for sequences, tags and lengths. For a signature, the raw size contains four byte for the signature number. As the ASN.1 encoding uses the *Distinguished Encoding Rules* (DER), the size of the encoded signature number as integer depends on its value. Therefore the size of an encoded signature will increase by a few bytes with later signatures.

5.1 Single Tree

A single tree suffers from high initialization costs, at least with large signature numbers. However, it needs less space for its state and has smaller signatures compared to tree chaining.

Therefore this version is analyzed with a small tree using $H = 10$ and a medium-sized tree using $H = 16$, which provide 1024 and 65536 signatures respectively. Both have $K = 4$, while w is chosen as 4 and 16 so that the parameters correspond to those used by Rohde et al.

The size of the single tree program executable is 10424 bytes. This includes the communication, operating system and verification code.

5.1.1 Performance

Table 5.1 shows the performance of my implementation on the cards. The small tree performs quite well with initialization times between 15 and 19 seconds. Even with $w = 16$, signing a 256-bit message is possible in 87 milliseconds on average and just above 100 at worst.

As expected with the medium-sized tree, initialization takes quite some time longer with 15 to 20 minutes. Theoretically, this effort is doubled each time the tree height is increased by one. This effect can be confirmed as the numbers are a factor of approximately 64 larger compared to the small tree, which corresponds to the increase in height of 6. The run times for signature generation, however, are still clearly below 200 milliseconds.

The time to initialize a large tree with $H = 20$ can be estimated to take 16 times longer than the middle one, which would already be more than four to five hours. Consequently, a single tree can be used only with a small number of possible signatures or when initialization times are not critical. The latter can be the case when keys are generated

Parameters				Initialization	Sign		
H	K	w	m		min	max	avg
10	4	4	32	14.6	0.026	0.086	0.075
10	4	16	32	18.8	0.022	0.100	0.086
16	4	4	32	925.4	0.043	0.134	0.095
16	4	16	32	1199.1	0.044	0.159	0.111

Table 5.1: Run times in seconds for a single tree with different parameters on hardware.

during personalization and not interactively. Another approach could be creating the keys on a more powerful system and transmitting them to the card inside of a secure environment.

Regarding the Winternitz parameter, it is interesting to see that the run times increase only by around 29% although w has been quadrupled.

Parameters				Initialization	Sign		
H	K	w	m		min	max	avg
10	4	4	32	7.7	0.021	0.044	0.039
10	4	16	32	9.6	0.019	0.048	0.042

Table 5.2: Run times in seconds for a single tree with different parameters in the simulator.

The results reported by the profiling function of the simulator are given in table 5.2. They are nearly a factor of two faster than in reality. Nevertheless, the actual computation of the simulation was much slower and took hours even with the small tree. Therefore the results for the medium-sized one are omitted here.

5.1.2 Memory Requirements

Parameters				State	Mask	Public Key		Signature	
H	K	w	m			Raw	ASN.1	Raw	ASN.1
10	4	4	32	1680	576	611	645	2292	2310
10	4	16	32	1648	544	579	613	1236	1254
16	4	4	32	2448	768	803	837	2388	2407
16	4	16	32	2416	736	771	805	1332	1351

Table 5.3: Sizes in bytes of state, public key and signature for a single tree using different parameters. The mask is included in both state and public key. The difference between raw and ASN.1 gives the encoding overhead.

Table 5.3 gives an overview of the memory requirements.

The state needs to be saved in NVM and includes the masks as well as one page for the control structure. Its size is defined mostly by the tree parameters H and K. Only the number of masks is a little smaller with the larger Winternitz parameter.

The masks are also included in the public key and account for its biggest part.

In terms of signature size, the Winternitz signature has the largest influence. The authentication path needs only H nodes, which means 160 and 256 bytes. The signature number is just a 4 byte integer. Therefore the difference between the Winternitz parameters 4 and 16 is significant, reducing the overall size by 42 to 46 percent.

As you can see, the difference between raw and ASN.1 encoded form is rather small. Public keys additionally include an algorithm identifier with about 10 byte in their encoding. The overhead for the encoding varies between 4 and 5 percent for public keys and only 0.8 to 1.5 percent for signatures.

This is due to the coarse ASN.1 definitions in section 4.1, where authentication paths, Winternitz signatures and masks are simple octet strings. If we had defined them more accurately as sequences of nodes, this would add two byte for tag and length to every node, resulting in a complete overhead of more than 13% for signatures.

5.1.3 Cycle Count Analysis

This section gives an estimate for the number of NVM writes issued during key and signature generation. These numbers can be combined to the overall cycle stress for the lifetime of a key pair. Here they are calculated for a single tree with $H = 20$ and $K = 8$. The Winternitz parameter has no influence in this case. For different parameter combinations one can simply adapt the numbers in table 5.4.

When generating the tree, the nodes needed for the structures are encountered at different positions. Hence every node is written separately and the number of write cycles corresponds to the number of nodes saved in the upper half of table 5.4.

In each signature step, the state control structure is modified, which fits into one page and therefore needs one cycle. Every second iteration a node is saved in Keep, resulting in 0.5 write cycles per step. The remaining structures are buffered and written together every step. Without authentication path, these are $4 \cdot (H - K) - 2 = 46$ nodes requiring 3 pages, which are modified every time by generating the next authentication path and distributing treehash updates.

As 3 pages can contain 48 nodes, they also include the first two nodes of the authentication path. Therefore a fourth page is written only if the authentication path is modified at more than two nodes and this happens only ever fourth time, resulting in additional 0.25 cycles per step[1].

Part	Structure	Size	Write Cycles
Initialization	NextSeed	12	12
	Auth	20	20
	TreehashNode	12	12
	Retain	247	247
			291
Per Step	State		1
	Keep	1	0.5
	Remain	66	3.25
			4.75

Table 5.4: Estimated NVM write cycles for a single tree with $H = 20$ and $K = 8$. The upper part is for initialization and the lower represents the average of all signature steps. The size is the number of nodes in a structure.

With an initialization part I and S cycles per step the overall number of cycles c is calculated as

$$c = I + 2^H \cdot S = 291 + 2^{20} \cdot 4.75 = 4981027.$$

These numbers show that the initialization part is not significant, even with a larger K. Therefore every node is written on its own during this process. Instead the modifications in the signature steps are optimized.

5.2 Tree Chaining

In order to get usable initialization times with an extensive number of possible signatures, one has to use tree chaining. The smart card implementation uses two levels. For simplicity, symmetrical trees are used here, meaning both levels have identical parameters H, K and w.

With 11396 byte, the size of the tree chaining program executable is not much larger than the other one.

With tree chaining, it does not make much sense to use few signature numbers. Therefore the analysis is done with a medium-sized tree using $H = 8$ and a large tree using $H = 10$. As both trees together have twice the height, we get a maximum number of 65536 and 1048576 signatures respectively.

For the medium-sized tree K is set to 2, for the large one to 4. As an attempt to compensate the increased signature size, we additionally test with a Winternitz parameter of 32. In order to get a better view on the parameter's influence on run times and signature sizes, $w = 8$ is also included closing the gap between 4 and 16.

[1] To be precise, the last two nodes again do not fit in this page, demanding an additional (fifth) write cycle every 2^{18} steps and thus a total of $\frac{1}{2^{18}} \cdot 2^{20} = 4$ cycles. This is however clearly negligible.

5.2.1 Performance

Parameters				Initialization	Sign		
H	K	w	m		min	max	avg
8	2	4	32	5.6	0.043	0.106	0.087
8	2	8	32	5.8	0.041	0.105	0.086
8	2	16	32	7.2	0.044	0.124	0.101
8	2	32	32	10.5	0.057	0.173	0.140
10	4	4	32	22.2	0.043	0.106	0.092
10	4	8	32	22.8	0.041	0.105	0.091
10	4	16	32	28.3	0.045	0.124	0.108
10	4	32	32	41.5	0.058	0.176	0.150

Table 5.5: Run times in seconds for tree chaining with different parameters on hardware.

Table 5.5 shows the run times on hardware. With $H = 8$ and $K = 2$, we have similar conditions like the single tree version using $H = 16$ and $K = 4$ above. Yet the initialization is 150 to 160 times faster and now possible within a few seconds, while signature generation times stay nearly the same, even drop a bit.

Tree chaining allows even the large key with a total height of 20 to be generated in less than 30 seconds for $w = 16$. Thereby the signature times increase only slightly compared to the mid-sized key. Even with $w = 32$ they take only 150 milliseconds on average.

Due to the distribution of updates between both levels, the worst case signature time in relation to its average is not much larger than with a single tree.

Again, the connection between Winternitz parameter and run times is interesting. While w is doubled each time, the initialization time is increasing faster. Between 4 and 8 it stays nearly the same, whereas it increases by 24% between 8 and 16 and by 47% between 16 and 32.

Parameters				Initialization	Sign		
H	K	w	m		min	max	avg
8	2	4	32	3.0	0.029	0.053	0.040
8	2	8	32	3.0	0.026	0.050	0.042
8	2	16	32	3.7	0.029	0.059	0.049
8	2	32	32	5.3	0.039	0.082	0.069
10	4	4	32	11.9	0.029	0.052	0.047
10	4	8	32	11.7	0.026	0.050	0.044
10	4	16	32	14.7	0.029	0.058	0.052
10	4	32	32	21.1	0.038	0.081	0.072

Table 5.6: Run times in seconds for tree chaining with different parameters in the simulator.

5.2.2 Memory Requirements

The main disadvantage of tree chaining is the increased signature size shown in table 5.7.

While the signatures for a single tree with $H = 16$ and $w = 4$ required 2388 byte, tree chaining adds another 1088 byte for the upper signature. This corresponds to a considerable increase of 45.6%. However, with $w = 8$ it can be nearly compensated as the signature size drops to 2436, which is only 2% more than with a single tree.

For $w = 16$, the single tree needs 1332 byte for a signature and tree chaining 560 byte more. With about 42.0%, this increase is close to the previous, but doubling w to 32 in this case reduces the size only to 1588, which still corresponds to a plus of 19.2%.

This shows that, similar to the run times in the previous section, the decrease in signature size drops each time the Winternitz parameter is doubled.

The state needs additional space for maintaining two trees plus pending tree. Most notably it needs to store the upper signature in NVM. In return, the number of masks is lower, as they can be used for both levels, resulting in a smaller public key.

Parameters				State	Mask	Public Key		Signature	
H	K	w	m			Raw	ASN.1	Raw	ASN.1
8	2	4	32	3760	512	549	589	3476	3505
8	2	8	32	3376	480	517	557	2436	2465
8	2	16	32	3200	480	517	557	1892	1921
8	2	32	32	3056	448	485	525	1588	1617
10	4	4	32	4304	576	613	653	3540	3569
10	4	8	32	3920	544	581	621	2500	2529
10	4	16	32	3744	544	581	621	1956	1985
10	4	32	32	3600	512	549	589	1652	1681

Table 5.7: Sizes in bytes of state, public key and signature for tree chaining using different parameters. The masks are included in both state and public key. The difference between raw and ASN.1 gives the encoding overhead.

The overhead for encodings is slightly higher because they contain more structure. Signatures have up to 1.8 percent, public keys between 6.5 and 8.3 percent.

5.2.3 Cycle Count Analysis

With tree chaining, the cycle count analysis is a bit more complicated. The following numbers give an estimate for an example with $H = 10$ and $K = 4$. As the upper Winternitz signature has to be stored and updated in NVM, its size and therefore the upper Winternitz parameter it depends on become relevant. In order to simplify things here, it is calculated with three pages or 768 byte. That is sufficient for $w \geq 8$.

Part	Structure	Size	Write Cycles
Initialization	NextSeed	6	6
	Auth	10	10
	TreehashNode	6	6
	Retain	11	11
			33
Per Refresh	Copy Lower Auth	10	1
	Copy Lower TreehashNode	6	1
	Copy Lower NextSeed	6	1
	Upper Keep	1	0.5
	Upper Remain	32	2
	Upper Signature	46	3
	Upper Updates	16	3
			11.5
Per Step	State		1
	Lower Keep	1	0.5
	Lower Remain	32	2
	Pending Stack	10	1
			4.5

Table 5.8: Estimated NVM write cycles for tree chaining with $H = 10$ and $K = 4$. The upper part is for initializing one tree, the middle for refreshing the lower tree and the last one represents an average of all signature steps.

The initialization cost remains in principle the same as with a single tree and is only adjusted to the parameters. It is however executed 2^H times for the lower and once for the upper tree.

The per signature step effort is also different. On the one hand the update structure now fits into two pages even with the complete authentication path. On the other hand the stack of the pending tree has to be written as it is modified in each step.

New is the refresh part which sums up all cycles for a refresh of the lower tree. Authentication Path, TreehashNode and NextSeed have to be copied from pending to lower tree which takes one write each. Furthermore, the authentication

path in the upper tree and the upper signature for the root of the lower tree have to be updated. As said before, the size of the latter is three pages.

Lastly, the updates for the upper tree have to be considered which are detached from the authentication path generation and instead distributed over all lower tree steps. In the worst case there are $(H-K)/2 = 3$ updates and they are all done separately. Because just the treehash and stack part is modified this results in 3 write cycles for each upper signature refresh.

Finally, with an initialization part I, R cycles per refresh and S per step the overall number of cycles c is calculated as

$$c = (2^H + 1) \cdot I + 2^H \cdot R + 2^{2H} \cdot S = 1025 \cdot 33 + 1024 \cdot 11.5 + 2^{20} \cdot 4.5 = 4764193.$$

Even though initializing and refreshing trees needs a lot more write cycles than with a single tree, they are still about two orders of magnitude below the costs of all signature steps. Consequently optimization was again focused on this part.

5.3 Verification

Parameters			Verification
H	w	m	
10	4	32	0.022
10	16	32	0.017
16	4	32	0.023
16	16	32	0.018

Table 5.9: Run times in seconds for the verification of a signature with different parameters on hardware.

With the Merkle scheme verifying a signature is generally easier than generating one because no updates have to be spent on future authentication nodes and seeds. The verification consists simply of one leaf node computation out of the verified Winternitz signature and the following reconstruction of the root.

As the run times in table 5.9 show, the first part dominates the complexity. Increasing the Winternitz parameter makes the verification even faster in this case. Increasing the height has in contrast little influence.

5.4 Side-Channel Resistance

Side-channel attacks aim at vulnerabilities in the implementation of a crypto scheme rather than the algorithm itself. That means all theoretical proofs are worthless, if the hardware or software it is running with leaks information about its secrets.

Some general classes of side-channel attacks are:

- **Timing Attacks** measure the time used for computations
- **Power Attacks** measure the power consumption of a hardware device during computations
- **Fault Attacks** inject errors into computations

In order to analyze possible side-channel attack szenarios, it is important to determine what the secrets in this scheme are and how they are used. The complete hash tree will be made public sooner or later, because every node is part of an authentication path at some point, the root is even placed in the public key. Instead, the secrets are the Winternitz secret keys and the seeds used to construct them.

Very popular is Differential Power Analysis (DPA) which requires that the same secret key is used with many different inputs. Those traces are then statistically analyzed.

Now the one-time keys in the Merkle scheme become an interesting characteristic. Every Winternitz key pair may be used only for one message and during Winternitz signature generation the AES key is changed in every iteration. Therefore it is impossible for an attacker to gain multiple traces with the same key. Finally, only one AES output for each Winternitz block is returned to the attacker.

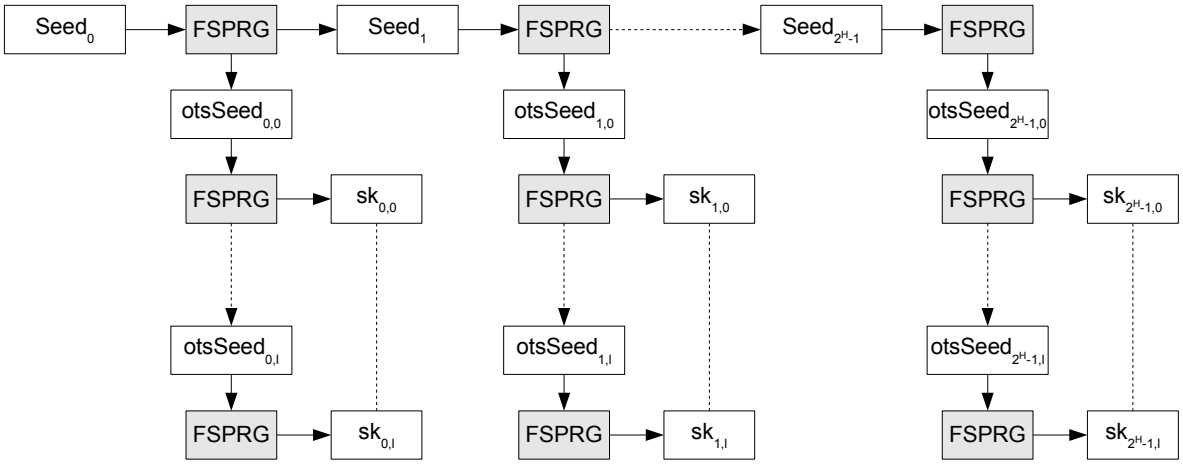


Figure 5.1: Modified construction of one-time keys with the FSPRG. Each of the one-time seeds is updated l times resulting in l data blocks, which are used as Winternitz key. The one-time seeds themselves are computed in the same way as before.

Sections 2.3.1 and 2.4.1 give two forms for the construction of one-time seeds. The forward secure has an additional advantage over the direct one, because the input seed is changed every time. The FSPRG constructs only two blocks with the same AES key, which should not be enough for a successful attack.

The other method uses the same seed as key for every one-time seed together with a sequence of known inputs (see the definition of the PRG in section 2.1.2). With a million signatures this would give a million different traces and make the scheme highly vulnerable to attacks like DPA.

In any way keys and seeds are generated multiple times. They are traversed once during initialization, once during the signature generation process and once by every treehash instance. Seeds are additionally passed through another time by NextSeed for each treehash height. This gives maybe a dozen iterations, depending on the parameters, which execute however the same computation every time. This means the traces are the same, but they can be combined to eliminate background noise and improve their quality.

Parameters				Initialization	Sign		
H	K	w	m		min	max	avg
8	2	8	32	6.3	0.043	0.112	0.093
8	2	16	32	7.6	0.047	0.130	0.106
10	4	8	32	24.6	0.044	0.112	0.098
10	4	16	32	29.8	0.047	0.129	0.113

Table 5.10: Run times in seconds for the tree chaining version with forward secure secret key generation.

The generation of Winternitz secret keys gives l traces out of the one-time seed. With good traces, a successful DPA sometimes requires no more than 100 and l can easily get into this region. On an insecure platform this might become a problem. To be on the safe side, the construction can be modified to use the same forward secure approach as with the seeds, where every key is used only twice. This modification is illustrated in figure 5.1.

Unfortunately, it needs double the AES operations for the secret key generation in return. The resulting performance impact depends on the Winternitz parameter, because a larger w means less data blocks and more iterations during public key generation, reducing the significance of secret key computation on the total complexity. With $w = 8$, the penalty is 7.2% and 8.6%. With $w = 16$, it shrinks to 2.1% and 2.7%.

With this construction, the scheme could be considered as secure against DPA as it provides no point of attack.

Fault attacks like Differential Fault Analysis (DFA) should also be hardly possible, because they require the attacker to gain at least one corrupted ciphertext together with the correct one. Again, as every Winternitz key is used only once, an attacker can get at most one output for each data block, corrupted or not.

6 Comparison

This chapter compares the performance and security level of this scheme with others.

In section 6.1 the results from the previous chapter are compared with those of the Merkle implementation by Rohde et al in [RED+08]. Please note that they use a different definition of the Winternitz parameter as the number of bits to be signed with one data block (like in [BBD08]), which is just a special case of our definition when it is a power of two. This means our w is two to the power of their w.

Section 6.2 shows performance figures for RSA and ECDSA as common signature algorithms. Both use the hardware acceleration provided by the SLE78.

Section 6.3 evaluates the security level of the Extended Merkle Signature Scheme and compares it to those of the previous ones.

6.1 Previous Implementation

As said before, the Merkle Signature Scheme has already been implemented on a smart card architecture. In [RED+08] Rohde et al achieve their run times with an 8-bit AVR Atmel microcontroller clocked at 16 MHZ. They also use a block cipher to construct the hash function and describe a version with AES hardware acceleration as well as a version with software AES. Table 6.1 shows both their results as given next to the corresponding run times for my implementation (see the previous chapter).

Parameters				Software AES		Hardware AES		My Implementation	
H	K	w	m	Sign	Verify	Sign	Verify	Sign	Verify
16	4	4	32	1.072	0.085	0.317	0.024	0.095	0.023
16	4	16	32	1.665	0.127	0.504	0.038	0.111	0.018

Table 6.1: Average run times in seconds for signature generation and verification with software and hardware accelerated AES as presented in [RED+08] compared to those of my implementation.

From the hardware point of view, my implementation has a clear advantage as it can rely on a 16-bit microprocessor with double the clock frequency.

But Rohde et al use Merkle in the classical version meaning they generate a leaf node by hashing the public key, which requires l AES operations plus one for the length padding. XMSS instead constructs leafs using a hash tree starting from the l nodes as described in section 2.4. A tree with height H has $2^{H+1} - 1$ nodes, where each node except the root is input to the hash function exactly one time. That makes

$$2^{H+1} - 2 \;=\; 2^{\log_2 l + 1} - 2 \;=\; 2l - 2$$

AES operations instead of $l + 1$.

On the other hand the hash function in my thesis requires only two AES operations instead of three, but has in return some additional expenses resulting from the input masking.

Parameters				Rohde et al			My Implementation		
H	K	w	m	State	Public Key	Signature	State	Public Key	Signature
16	4	4	32	1472	16	2386	2448	803	2388
16	4	16	32	1472	16	1330	2416	771	1332

Table 6.2: Sizes in bytes of state, public key and signatures as presented in [RED+08] compared to those of my implementation.

Altogether, my implementation is three times faster with $w = 4$ and four times with $w = 16$ regarding the average signature generation time.

The initialization times in the previous sections show that this implementation, in contrast to Rohde et al, has the ability to generate keys on the smart cards. Small trees can be handled directly with the single tree version. However, far more signatures are possible with the tree chaining extension. The increase in signature size is acceptable when using larger Winternitz parameters in turn.

Table 6.2 shows the size of keys and signatures in the version of Rohde et al. They use the Merkle scheme with predefined parameters. Therefore their state is just the sum of the necessary structures without any overhead.

My implementation adds a control structure, using one page with 256 byte, and the masks needed for XMSS. Those masks are also responsible for the large public key sizes compared to Rohde et al, who need only 16 bytes for the root.

The signature sizes are the same with the only difference being that they use two bytes for the number whereas my implementation uses four.

6.2 RSA and ECDSA

Algorithm	Sign	Verify	Private Key	Public Key	Signature
RSA 1024	0.070	0.003	256	132	128
RSA 2048	0.190	0.007	512	260	256
ECDSA 256	0.100	0.058	32	32	32

Table 6.3: Figures for RSA and ECDSA on the SLE78. The left side shows the run times in seconds for signature generation and verification. The right side shows sizes in bytes for keys and signatures.

As said in section 3.3, the SLE78 offers a coprocessor for asymmetric cryptography. Table 6.3 shows the performance for signature generation and verification with RSA and ECDSA on this smart card using the hardware acceleration. It should however be noted that these numbers include countermeasures against side-channel attacks in software. This slows down the run times for asymmetric cryptography, unlike the SCP where they are implemented in hardware.

RSA uses the small standard public key exponent $e = F_4$, which is the reason for its fast verification times.

The time needed to find large prime numbers on the card has a large statistical variation. On average, key generation for RSA takes about 1 second with 1024 bit and 11 seconds with 2048 bit modulus.

The results in the previous chapter show that this implementation has run times for signature generation comparable to those of RSA and ECDSA. Initialization is also possible within some seconds, at least with the tree chaining version. However, this depends highly on the number of possible signatures.

One of the major drawbacks of the Merkle scheme are its memory requirements. Signatures and keys are much larger than those of RSA. Elliptic curve cryptography needs even less space. In this domain XMSS cannot keep up.

6.3 Security Level

The security level is used to indicate the strength of a cryptographic scheme. It is always based on the best known attack on the algorithm.

The level is represented in security bits, which estimate the computational effort required to break the scheme. A security level of n bit means that a successful attack needs on average about 2^n operations.

Security Level	Symmetric Key	Finite Field	Integer Factorization	Elliptic Curve
80	2TDEA	$L = 1024, N = 160$	$k = 1024$	$f = 160 - 223$
112	3TDEA	$L = 2048, N = 224$	$k = 2048$	$f = 224 - 255$
128	AES128	$L = 3072, N = 256$	$k = 3072$	$f = 256 - 383$
192	AES192	$L = 7680, N = 384$	$k = 7680$	$f = 384 - 511$
256	AES256	$L = 15360, N = 511$	$k = 15360$	$f = 512+$

Table 6.4: The security level in bit of symmetric key algorithms compared to key sizes in Finite Field Cryptography (FFC), Integer Factorization Cryptography (IFC) and Elliptic Curve Cryptography (ECC). Source: [NIS07]

Table 6.4 shows a comparison of the security level of cryptographic algorithm classes as proposed by the National Institute of Standards and Technology (NIST) in section 5.6.1 of its Special Publication 800-57 “Recommendation for Key Management - Part 1” [NIS07].

It contains the following classes:

- Symmetric Key gives a block cipher with the indicated strength. 2TDEA and 3TDEA stand for 2-key and 3-key tripe-DES.
- Finite Field Cryptography refers to algorithms that use the discrete logarithm problem, for example the Digitial Signature Algorithm (DSA) and Diffie-Hellmann (DH) key agreement. The value L is the size of the field and N is the size of the subgroup.
- Integer Factorization Cryptography includes the most popular RSA algorithm. Its key size corresponds to k, the size of the modulus n.
- In Elliptic Curve Cryptography the key size is equivalent to f, the order of the base point.

Rohde et al specify the security level of their scheme in [RED+08] with 128 bit.

A single Winternitz signature has l data blocks and in order to forge a new signature in a chosen-message attack one needs to invert only one of them. In the same way, any of the 2^H signatures in the Merkle tree can be attacked. This means the success probability for a random preimage search is at least

$$\frac{2^H \cdot l}{2^n} = \frac{2^{H+\log_2(l)}}{2^n} = \frac{1}{2^{n-H-\log_2(l)}}$$

which corresponds to an upper bound for the security level of

$$b \leq n - H - \log_2(l)$$

bit.

Parameters				Security Level
H	K	w	m	
10	4	4	32	92
10	4	16	32	78
16	4	4	32	86
16	4	16	32	72

Table 6.5: The provable security level of the presented scheme with a single tree. The output length n is 128 bit.

This is however not the provable security level, because the PRF we use is not a perfect permutation. There might be multiple AES128 keys encrypting one input to the same ciphertext. If there is more than one preimage for a Winternitz block, then there are multiple paths to each final data block in the public key and an attacker needs to find only one of them.

Parameters				Security Level		
H	K	w	m	Lower Tree	Upper Tree	Complete
8	2	4	32	94	96	85
8	2	8	32	90	91	81
8	2	16	32	80	82	71
8	2	32	32	63	65	54
10	4	4	32	92	94	81
10	4	8	32	88	89	77
10	4	16	32	78	80	67
10	4	32	32	61	63	50

Table 6.6: The provable security level of the presented scheme with tree chaining. The output length n is 128 bit.

In [BH11] the authors analyze the security level of XMSS during their proof in section 3. They give a lower bound of

$$b \geq n - H - 3 - w - 2\log_2(lw)$$

bit for a single tree. For the parameters from section 5.1 this results in the values shown in table 6.5.

With tree chaining, the security level of the complete scheme can be estimated downward with

$$b_{complete} \geq \min\left\{b_{upper},\ b_{lower} - H\right\} - 1$$

bit, leading to the results in table 6.6 for the parameter combinations in section 5.2.

7 Conclusion

At the end of my thesis, I would like to draw a conclusion and point out the pros and cons of the presented work. Finally, section 7.1 shows some points, where it could be continued.

In the introduction I gave a motivation and an overview of what I wanted to achieve in this thesis. Chapter 2 presented the fundamentals of hash-based cryptography in form of the Winternitz and Merkle scheme and their prerequisites. It continued with a description of the extended Merkle scheme, which lowers the requirements on the hash function, and tree chaining used to speed-up the key generation. Chapter 3 was about smart cards and their interface followed by a presentation of the implementation in chapter 4. In chapter 5 the performance is analyzed including run times and memory requirements. Finally, the security level is assessed in chapter 6 and this scheme is compared to others.

These results show that the Extended Merkle Signature Scheme can be implemented with key generation on a smart card. The performance for signature generation is comparable to RSA with hardware support (see section 6.2).

Overall, the implementation meets the goals specified in the introduction and can be used for all three applications. Section 5.2 shows that signature generation is well under one second and key generation takes less than a minute with one million possible signatures, using the tree chaining version. The large signature numbers allow to use the presented scheme even for authentication purposes. One million signatures and a key lifetime of five years result in more than 500 available signatures per day. Regarding document signing, it further benefits from the addition of forward security in section 2.4.1. For the third application presented in the introduction, code signing, this scheme seems perfectly suitable, because the time-consuming key and signature generation are handled outside. Code signing needs only signature verification on the smart card, which is very fast according to the run times in section 5.3.

Compared to the previous Merkle implementation of Rohde et al in [RED+08], section 6.1 shows that this implementation features not only faster run times with more available signatures, but also a modified scheme with a more accurate security proof and security level assessment. The most important difference however is the possibility to generate keys on the smart card using tree chaining.

One of the advantages of the Merkle scheme is its ability to share the hardware acceleration with symmetric key cryptography. This eliminates the need for another expensive coprocessor like the Crypto@2304T and allows the scheme to run with the same speed on a cheaper smart card, for example the SLE77. The only hardware requirement is a block cipher and additionally a random number generator if one wishes to generate keys.

If we consider a pure software implementation, the Merkle scheme is clearly ahead of the classical RSA algorithm, whose performance suffers drastically without proper hardware acceleration. In their paper, Rohde et al notice that Merkle loses only approximately a factor of three with an optimized software AES implementation compared to the version with hardware AES.

A general drawback with the Merkle scheme is the size of signatures and the state. The extended version additionally enlarges the public key. On the other hand, it allows to use a second-preimage resistant hash function in the tree. This halves the size of the authentication path compared to the classic version, which requires collision resistance for the same security level. The signature size can be further reduced with larger Winternitz parameters, which however decrease both performance and security level. Compared to RSA, the signature size of XMSS is clearly higher. But regarding today's amounts of data and communication speeds, this is not significant and therefore no problem at all.

Another disadvantage is that the state is modified in each step, resulting in a lot of NVM cycle stress compared to standard signature algorithms with a constant private key. However, the cycle count numbers in chapter 5 are under five million even with large keys. As a single sector allows about 16.5 million write cycles (see section 3.3.3), this should be more than enough for the lifetime of a smart card. Otherwise the next key could be placed in a different sector.

7.1 Future Work

The following sections suggest some improvements for the presented work.

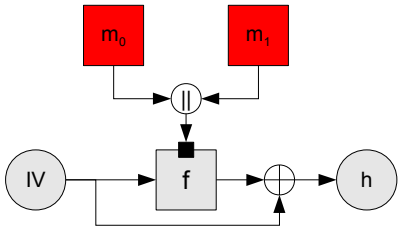


Figure 7.1: Davies-Meyer compression function with AES256. The left and right child nodes are concatenated and used as 256-bit key to encrypt the chain input. As this needs only one iteration, the input is the initialization vector and the output the final hash value. Therefore the XOR could be left out, because it can be inverted by anyone.

7.1.1 Communication

So far, the implementation focussed on the scheme itself. A real world application would need more communication code.

This includes Protocol Type Selection (see section 3.2.3) for high communication speeds with any PCSC reader and a T=1 protocol implementation.

Furthermore, the SLE78 features *Peripheral Event Channel* (PEC) controllers to handle data copies from and to various locations in parallel to normal program operation. This includes the UART as possible target allowing asynchronous communication.

This can be used to send out the first part of the signature while updating the authentication path and accordingly hide some of the communication time.

7.1.2 AES256 Hashing

The hash function in section 2.1.4 gets a 256-bit input it maps on a 128-bit hash value. This is done with two iterations of AES128, where each time 128-bit are used as key.

With the Davies-Meyer compression function and AES256 the concatenated input could be used as key in one iteration like shown in figure 7.1.

Parameters				Initialization	Sign		
H	K	w	m		min	max	avg
8	2	8	32	5.0	0.039	0.095	0.079
8	2	16	32	6.7	0.043	0.116	0.096
10	4	8	32	19.8	0.040	0.095	0.083
10	4	16	32	26.4	0.043	0.117	0.102

Table 7.1: Run times in seconds for the tree chaining version with AES256 hash function.

The run times in table 7.1 show a performance speed-up on key generation of approximately 15.5% with $w = 8$ and 7.3% with $w = 16$, compared to the results in section 5.2. A larger Winternitz parameter increases the percentage of the public key generation, which does not profit from a faster hash function, on the total complexity and therefore reduces its advantage. The average signature generation is about 9.2% and 5.5% faster.

However, before using this construction, its second-preimage resistance has to be analyzed. Second-preimage means another key that maps the same input onto the same output has to be found. For one fixed input there are 2^{256} keys and 2^{128} possible outputs. Therefore on average, 2^{128} keys map that input onto the same output. With guessing one gets a success probability of

$$\frac{2^{128}}{2^{256}} = \frac{1}{2^{128}}$$

which corresponds to a security level of 128 bit. But there might be more efficient ways to find them.

7.1.3 Double-Block-Length Constructions

AES supports only a block length of 128 bit. In order to achieve a security level of 128 bit and more, it could be used in a double-block-length construction like Hirose [Hir06]. This would probably increase run times and memory requirements by a factor of two.

7.1.4 Parameter Combinations

The signature scheme has four parameters, which determine the number of possible signatures, the run times as well as key and signature sizes. The smart card implementation allows to choose them freely according to the environmental conditions. For example, if a lot of NVM space is available, one can choose a larger retain height K and consequently get faster signature times.

There is some optimization space regarding the Winternitz parameter w. On the one hand it increases the length of the hash chain for each data block linearly, as it is $w - 1$. On the other hand it reduces their number l. This decrease is however not linear but logarithmic (see the definitions in section 2.2).

The number of operations needed to generate a Winternitz public key corresponds to the number of data blocks times the length of the hash chain. Therefore run times can be optimized by minimizing the function

$$(w - 1) \cdot l(w, m)$$

for a given m.

Table 5.5 shows the correlation between performance and Winternitz parameter. In each step, w is doubled but the initialization time increases by 3% between 4 and 8, whereas it increases by 27% between 8 and 16 and by 45% between 16 and 32.

l also defines the size of the Winternitz signature, which is the biggest part of the Merkle signature. This is shown in table 5.7. It could make sense to reduce this at the cost of some additional computational effort. The decline in signature size, however, drops with every step.

The costs in both performance and space can be reduced with smaller messages. Therefore m should be ideally chosen twice as large as the security level of the scheme so that it provides the same level of resistivity against collision attacks.

Finding optimal parameter combinations could be investigated further.

7.1.5 Side-Channel Resistance

Section 5.4 briefly sketches possible side-channel attack szenarios and attempts to analyze the resistivity of the provided scheme. This topic needs, however, more attention.

Acknowledgement

I would like to thank Prof. Dr. Johannes Buchmann and Andreas Hülsing for supervising my thesis.

My thanks also go to Infineon Technologies, not only for providing the necessary hardware, software and documentation, but also for giving me the opportunity to come to Munich. I am especially greatful to Wieland Fischer and Olaf Brixel, who were helping me with all my questions and problems.

Finally, I would like to thank Danusia and Simon for proofreading this work.

Bibliography

[And02] Ross Anderson. Two Remarks on Public Key Cryptology. Technical Report UCAM-CL-TR-549, University of Cambridge, Computer Laboratory, December 2002.

[BBD08] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post Quantum Cryptography*, chapter Hash-based Digital Signature Schemes. Springer Publishing Company, Incorporated, 1st edition, 2008.

[BDE+11] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the Security of the Winternitz One-Time Signature Scheme. In *Africacrypt 2011*, Apr 2011. accepted.

[BDK+07] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Vuillau Camille. Merkle Signatures with Virtually Unlimited Signature Capacity. In *Proceedings of the 5th international conference on Applied Cryptography and Network Security*, ACNS '07, pages 31–45, Berlin, Heidelberg, 2007. Springer-Verlag.

[BDS08] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle Tree Traversal Revisited. In *Proceedings of the 2nd International Workshop on Post-Quantum Cryptography*, PQCrypto '08, pages 63–78, Berlin, Heidelberg, 2008. Springer-Verlag.

[BGD+06] Johannes Buchmann, Luis Carlos Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS - An Improved Merkle Signature Scheme. In *INDOCRYPT*, pages 349–363, 2006.

[BH11] Johannes Buchmann and Andreas Hülsing. XMSS - A Practical Forward Secure Signature Scheme based on Minimal Security Assumptions. In *Post-Quantum Cryptography - Proceedings of PQCrypto 2011*, 2011. accepted.

[BM99] Mihir Bellare and Sara K. Miner. A Forward-Secure Digital Signature Scheme. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 431–448, London, UK, UK, 1999. Springer-Verlag.

[DOTC08] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Vuillau Camille. Digital Signatures Out of Second-Preimage Resistant Hash Functions. In *Proceedings of the 2nd International Workshop on Post-Quantum Cryptography*, PQCrypto '08, pages 109–123, Berlin, Heidelberg, 2008. Springer-Verlag.

[GB08] S. Goldwasser and M. Bellare. *Lecture Notes on Cryptography*. Massachusetts Institute of Technology, 1996–2008. http://cseweb.ucsd.edu/~mihir/papers/gb.html.

[Hir06] Shoichi Hirose. Some Plausible Constructions of Double-Block-Length Hash Functions. In *Proceedings of the 13th international conference on Fast Software Encryption*, FSE'06, pages 210–225, Berlin, Heidelberg, 2006. Springer-Verlag.

[Inf10] Infineon. *SLx 70 Family Hardware Reference Manual*, November 2010.

[Inf11] Infineon. *SLx 70 Family Programmer's Reference Manual*, August 2011.

[ISOa] ISO/IEC 7816-3. *Identification cards - Integrated circuit cards - Part 3: Cards with contacts - Electrical interface and transmission protocols*.

[ISOb] ISO/IEC 7816-4. *Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange*.

[Mer89] Ralph C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[NIS07] NIST. *Special Publication 800-57: Recommendation for Key Management - Part 1*, March 2007.

[RED+08] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. Fast Hash-Based Signatures on Constrained Devices. In *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 104–117, Berlin, Heidelberg, 2008. Springer-Verlag.

[Sho96] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26:1484–1509, 1996.

[SK] Werner Schindler and Wolfgang Killmann. *AIS 31: Functionality classes and evaluation methodology for true (physical) random number generators*. Bundesamt für Sicherheit in der Informationstechnik (BSI).