Forward Secure Signatures on Smart Cards*

Andreas Hülsing, Christoph Busold, and Johannes Buchmann

Cryptography and Computeralgebra
Department of Computer Science
TU Darmstadt, Germany
{huelsing,buchmann}@cdc.informatik.tu-darmstadt.de,
christoph.busold@cased.de

Abstract. We introduce the forward secure signature scheme XMSS$^+$ and present an implementation for smart cards. It is based on the hash-based signature scheme XMSS. In contrast to the only previous implementation of a hash-based signature scheme on smart cards by Rohde et al., we solve the problem of on-card key generation. Compared to XMSS, we reduce the key generation time from $\mathcal{O}(n)$ to $\mathcal{O}(\sqrt{n})$, where n is the number of signatures that can be created with one key pair. To the best of our knowledge this is the first implementation of a forward secure signature scheme and the first full implementation of a hash-based signature scheme on smart cards. The resulting runtimes are comparable to those of RSA and ECDSA on the same device. This shows the practicality of forward secure signature schemes, even on constrained devices.

Keywords: forward secure signatures, smart cards, implementation, hash-based signatures.

1 Introduction

In 1997 Anderson introduced the idea of forward secure signature schemes (FSS) [3]. The idea behind FSS is the following: Even in the case of a key compromise, all signatures issued before the compromise should remain valid. This is an important property for all use cases where signatures have to stay valid for more than a short time period, including use cases like document signing or certificates. If for example a contract is signed, it is important that the signature stays valid for at least as long as the contract has some relevance. The solutions used today require the use of time stamps [13, 14]. This introduces the requirement for a trusted third party and the overhead of requesting a time stamp for each signature. FSS in turn already provide this property and thereby abandon the need for time stamps. To fulfill the forward security property, a signature scheme has to be *key evolving*, meaning, the private key changes over time. The lifetime of a key pair is divided into time periods. While the public key stays the same, the secret key is updated at the end of each time period. So far, it was shown

* Supported by grant no. BU 630/19-1 of the German Research Foundation (www.dfg.de).

that FSS can be efficiently implemented on PCs [6, 11]. As for common signature schemes, to be usable in practice, FSS must be efficiently implementable on smart cards. This is even more important in the case of FSS, as it has to be ensured that the secret key is updated and the former secret key is deleted. So far there exists no implementation of FSS on smart cards.

A candidate FSS is the hash-based FSS XMSS [6] because of its strong security guarantees (see Section 2). Moreover, XMSS benefits from hardware acceleration for block ciphers, which is provided by many smart cards. A severe problem of most FSS, including XMSS, is the costly key generation. XMSS key generation requires time linear in the number of signatures that can be generated using the same key pair. While this might be tolerable on PCs, it makes key generation on smart cards impractical. The only existing implementation of a hash-based signature scheme on smart cards [22] does not include on-card key generation for this reason. But on-card key generation is necessary for most use cases that benefit from the forward security property. I.e. to guarantee non-repudiation in the case of document signing, a signature key pair has to be generated on the smart card and must never leave this secure environment.

Our contribution. In this paper we introduce XMSS$^+$ which is based on XMSS and present an implementation on an Infineon SLE78 smart card. While the strong security guarantees of XMSS are preserved, XMSS$^+$ key generation requires only time $\mathcal{O}(\sqrt{n})$, for a key pair, that can be used to sign n messages. Thereby we make on-card key generation practical. This means we present the first implementation of a forward-secure signature scheme on a smart card. At the same time, it is the first full (including key generation) smart card implementation of a hash-based signature scheme. To achieve this, we use the tree chaining technique [9] and improve the idea of distributed signature generation [7]. To improve the performance, we implemented all used (hash) function families based on AES and exploit the hardware acceleration provided by the card. Using our implementation, the generation of a key pair, that can be used to generate 2^{20} signatures, can be done in $22.2s$. For such a key pair, signature generation took less than $106ms$, verification took no more than $44ms$. These timings are of the same order of magnitude than the runtimes for RSA and ECDSA on the same card using the asymmetric crypto co-processor.

Organization. We start with a description of XMSS in Section 2. XMSS$^+$, that enables key generation, is presented and analyzed in Section 3. We describe our implementation and present parameters and runtimes in Section 4. Finally, we give a conclusion in Section 5.

2 The eXtended Merkle Signature Scheme XMSS

In this section we describe the FSS XMSS [6]. While there exist many proposals for FSS, including [1, 2, 4, 10, 15–17, 19, 23], XMSS is the only FSS where the forward security is based on minimal security assumptions. XMSS uses a function family $\mathcal{F}$ and a hash function family $\mathcal{H}$. It is provably forward secure in the standard model, if $\mathcal{F}$ is pseudorandom and $\mathcal{H}$ second preimage resistant. As current research suggests that these properties are not threatened by the existence of quantum computers, XMSS$^+$ is assumed to be resistant against quantum computer based attacks. We first give a high level overview. XMSS is build on a one-time signature scheme (OTS), a signature scheme where a key pair can only be used once. To obtain a many-time signature scheme, many OTS key pairs are used and their public keys are authenticated using a Merkle Tree. A Merkle Tree is a binary hash tree. The leaves of the tree are the hash values of the OTS public keys. The root of the Merkle Tree is the XMSS public key. To overcome the need of storing all OTS key pairs, they are generated using a pseudorandom generator (PRG). We start the detailed description with the parameters used by XMSS, afterwards we give a description of the building blocks, namely, the Winternitz-OTS, the XMSS Tree, the leaf construction, and the PRG. Then we describe the algorithms for key generation, signature generation and verification. In the following we write log for $\log_2$ and $x \xleftarrow{\$} X$ if the value x is chosen uniformly at random from the set X.

Parameters. For security parameter $n \in \mathbb{N}$, XMSS uses a pseudorandom function family $\mathcal{F}_n = \{\mathrm{F}_K : \{0,1\}^n \rightarrow \{0,1\}^n | K \in \{0,1\}^n\}$, and a second preimage resistant hash function H, chosen uniformly at random from the family $\mathcal{H}_n = \{\mathrm{H}_K : \{0,1\}^{2n} \rightarrow \{0,1\}^n | K \in \{0,1\}^n\}$. It is parameterized by the message length $m \in \mathbb{N}$, the tree height $h \in \mathbb{N}$, the BDS parameter $k \in \mathbb{N}, k < h, k-h$ is even, and the Winternitz parameter $w \in \mathbb{N}, w > 1$. XMSS can be used to sign 2^h message digests of m bits. The Winternitz parameter w allows for a trade off between signature generation time and signature size. The BDS parameter k allows for a time-memory trade-off for the signature generation. Those parameters are publicly known.

Winternitz OTS. XMSS uses the Winternitz-OTS (W-OTS) from [5]. W-OTS uses the function family $\mathcal{F}_n$ and a value $X \in \{0,1\}^n$ that is chosen during XMSS key generation. For $K, X \in \{0,1\}^n$, $e \in \mathbb{N}$, and $\mathrm{F}_K \in \mathcal{F}_n$ we define $\mathrm{F}^e_K(X)$ as follows. We set $\mathrm{F}^0_K(X) = K$ and for $e > 0$ we define $K' = \mathrm{F}^{e-1}_K(X)$ and $\mathrm{F}^e_K(X) = \mathrm{F}_{K'}(X)$. Also, define

$$\ell_1 = \left\lceil \frac{m}{\log(w)} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{\log(w)} \right\rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

The secret signature key of W-OTS consists of ℓ n-bit strings sk_i, $1 \leq i \leq \ell$. The generation of the sk_i will be explained later. The public verification key is computed as

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) = (\mathrm{F}^{w-1}_{\mathsf{sk}_1}(X), \ldots, \mathrm{F}^{w-1}_{\mathsf{sk}_\ell}(X)),$$

with F^{w-1} as defined above. W-OTS signs messages of binary length m. They are processed in base w representation. They are of the form $M = (M_1 \ldots M_{\ell_1})$,

$M_i \in \{0, \ldots, w-1\}$. The checksum $C = \sum_{i=1}^{\ell_1}(w-1-M_i)$ in base w representation is appended to M. It is of length ℓ_2. The result is a sequence of ℓ base w numbers, denoted by $(T_1, \ldots, T_\ell)$. The signature of M is

$$\sigma = (\sigma_1, \ldots, \sigma_\ell) = (\mathrm{F}^{T_1}_{\mathsf{sk}_1}(X), \ldots, \mathrm{F}^{T_\ell}_{\mathsf{sk}_\ell}(X)).$$

It is verified by constructing $(T_1 \ldots, T_\ell)$ and checking

$$(\mathrm{F}^{w-1-T_1}_{\sigma_1}(X), \ldots, \mathrm{F}^{w-1-T_\ell}_{\sigma_\ell}(X)) \overset{?}{=} (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell).$$

The sizes of signature, public, and secret key are ℓn bits. For more detailed information see [5].

XMSS Tree. The XMSS tree utilizes the hash function H. The XMSS tree is a binary tree of height h. It has $h+1$ levels. The leaves are on level 0. The root is on level h. The nodes on level j, $0 \leq j \leq h$, are denoted by $N_{i,j}$, $0 \leq i < 2^{h-j}$. To construct the tree, h bit masks $B_j \in \{0,1\}^{2n}$, $0 < j \leq h$, are used. $N_{i,j}$, for $0 < j \leq h$, is computed as

$$N_{i,j} = \mathrm{H}((N_{2i,j-1} || N_{2i+1,j-1}) \oplus B_j).$$

Leaf Construction. The leaves of the XMSS tree are the hash values of the W-OTS public keys. To avoid the need of a collision resistant hash function, another XMSS tree is used to construct the leaves. It is called L-tree. The ℓ leaves of an L-tree are the ℓ bit strings $(\mathsf{pk}_0, \ldots, \mathsf{pk}_\ell)$ from the corresponding verification key. As ℓ is not necessarily a power of 2, there might not be sufficiently many leaves to get a complete binary tree. Therefore the construction is modified. A left node that has no right sibling is lifted to a higher level of the L-tree until it becomes the right sibling of another node. In this construction, the same hash function as above but new bitmasks are used. The bitmasks are the same for all L-trees. As L-trees have height $\lceil \log \ell \rceil$, additional $\lceil \log \ell \rceil$ bitmasks are required.

Pseudorandom Generator. The W-OTS key pairs are generated using two pseudorandom generators (PRG). The stateful forward secure PRG FsGen : $\{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$ is used to generate one seed value per W-OTS keypair, using the function family $\mathcal{F}_n$. Then the seed is expanded to the ℓ W-OTS secret key bit strings using $\mathcal{F}_n$. FsGen starts from a uniformly random state $S_0 \xleftarrow{\$} \{0,1\}^n$. On input of a state S_i, FsGen generates a new state S_{i+1} and a pseudorandom output R_i:

$$\textsc{FsGen}(S_i) = (S_{i+1} || R_i) = (\mathrm{F}_{S_i}(0) || \mathrm{F}_{S_i}(1)).$$

The output R_i is used to generate the ith W-OTS secret key $(\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$:

$$\mathsf{sk}_j = \mathrm{F}_{R_i}(j-1), 1 \leq j \leq \ell.$$

Key Generation. The key generation algorithm takes as input all of the above parameters. Then the whole XMSS Tree has to be constructed to obtain the value of the root node. We now detail this procedure. First, the bitmasks $(B_1, \ldots, B_{h+\lceil \log \ell \rceil})$ and the value X are chosen uniformly at random. Then, the initial state of FsGen, S_0 is chosen uniformly at random and a copy of it is stored as part of the secret key SK. The tree is constructed using the TreeHash algorithm, listed as Algorithm 1 below. Starting with an empty stack Stack and S_0, all 2^h leaves are successively generated and used as input to the TreeHash algorithm to update Stack. This is done by evaluating FsGen on the current state S_i, obtaining R_i and replacing S_i with S_{i+1}. Then R_i is used to compute the W-OTS public key, which in turn is used to compute the corresponding leaf using an L-tree. The leaf and the current Stack are then used as input for the TreeHash algorithm to obtain an updated Stack. The W-OTS key pair and R_i are deleted. After all 2^h leaves were processed by TreeHash, the only value on Stack is the root of the tree, which is stored in the public key PK.

Algorithm 1. TreeHash

```
Input: Stack Stack, node $N_1$
Output: Updated stack Stack

1. While top node on Stack has same height as $N_1$ do
   (a) $t \leftarrow N_1.height() + 1$
   (b) $N_1 \leftarrow \mathrm{H}((\mathsf{Stack}.pop() || N_1) \oplus B_t)$
2. $\mathsf{Stack}.push(N_1)$
3. Return Stack
```

The XMSS signature generation algorithm uses as subroutine the BDS algorithm [8] that is explained there. The BDS algorithm uses a state $\mathsf{State}_{\mathrm{BDS}}$ which is initialized during the above computation of the root. For details see [8]. The initial XMSS secret key $\mathsf{SK} = (S_0, \mathsf{State}_{\mathrm{BDS}})$ contains the initial states of FsGen and the BDS algorithm. The XMSS public key consists of the bitmasks $(B_1, \ldots, B_{h+\lceil \log \ell \rceil})$, the value X, and the root of the tree. As shown in [6], key generation requires $2^h(\ell + 1)$ evaluations of H and $2^h(2 + \ell(w + 1))$ evaluations of functions from $\mathcal{F}_n$.

Signature Generation. The signature generation algorithm takes as input a message M, the secret key SK and the index i. It outputs an updated secret key SK' and a signature Σ on the message M. To sign the ith message (we start counting from 0), the ith W-OTS key pair is used. The signature $\Sigma = (i, \sigma, \mathsf{Auth})$ contains the index i, the W-OTS signature σ, and the authentication path for the leaf $N_{0,i}$. The authentication path is the sequence $\mathsf{Auth} = (\mathsf{Auth}_0, \ldots, \mathsf{Auth}_{h-1})$ of the siblings of all nodes on the path from $N_{0,i}$ to the root. Figure 1 shows the authentication path for leaf i. We now explain how a signature is generated. On input of the ith message, SK contains the ith state S_i of FsGen. So, FsGen is evaluated on S_i to obtain S_{i+1}, which becomes the updated secret key, and R_i. R_i is used to generate the ith W-OTS secret key, which in turn is used to generate the one-time signature σ on M. Then the authentication path is computed using the BDS tree traversal algorithm from [8] which we explain next.

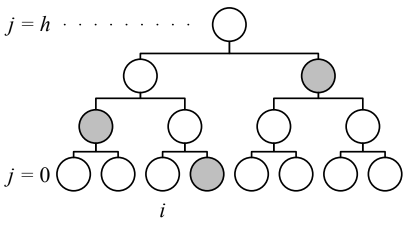

Fig. 1. The authentication path for leaf i

The BDS algorithm uses TreeHash to compute the nodes of the authentication path. The computation of a node on level i takes 2^i leaf computations and 2^i evaluations of TreeHash. If all this computation is done when the authentication path is needed, the computation of an authentication path requires $2^h - 1$ leaf computations and evaluations of TreeHash in the worst case. The BDS algorithm reduces the worst case signing time to $(h - k)/2$ leaf computations and evaluations of TreeHash. More specifically, the BDS algorithm does three things. First, it uses the fact that a node that is a left child can be computed from values that occurred in an authentication path before, spending only one evaluation of H. Second, it stores the right nodes from the top k levels of the tree during key generation. So these nodes, that are most expensive to compute, do not have to be computed again during signature generation. Third, it distributes the computations for right child nodes among previous signature generations. This is done, using one instance of TreeHash per tree level. The computation of the next right node on a level starts, when the last computed right node becomes part of the authentication path. The BDS algorithm uses a state $\mathsf{State}_{\mathrm{BDS}}$ of $2(h-k)$ states of FsGen and at most $\left(3h + \left\lfloor \frac{h}{2} \right\rfloor - 3k - 2 + 2^k\right)$ tree nodes. $\mathsf{State}_{\mathrm{BDS}}$ is initialized during key generation. After initialization, it contains the right nodes on the k top levels, the first authentication path (for $N_{0,0}$) and the second right node on each level. To compute the authentication paths, the BDS algorithm spends only $(h - k)/2$ leaf computations and evaluations of TreeHash to update its state per signature. This update is done such that at the end of the ith signature generation, $\mathsf{State}_{\mathrm{BDS}}$ already contains the authentication path for leaf $i + 1$. For more details see [8].

Signature Verification. The signature verification algorithm takes as input a signature $\Sigma = (i, \sigma, \mathsf{Auth})$, the message M and the XMSS public key PK. To verify the signature, the values $(T_0, \ldots, T_\ell)$ are computed as described in the

W-OTS signature generation, using M. Then the ith verification key is computed using the formula

$$(\mathsf{pk}_1, \dots, \mathsf{pk}_\ell) = (\mathrm{F}_{\sigma_1}^{w-1-T_1}(X), \dots, \mathrm{F}_{\sigma_\ell}^{w-1-T_\ell}(X)).$$

The corresponding leaf $N_{0,i}$ of the XMSS tree is constructed using an L-tree. This leaf and the authentication path are used to compute the path $(P_0, \dots, P_h)$ to the root of the XMSS tree, where $P_0 = N_{0,i}$ and

$$P_j = \begin{cases} \mathrm{H}((P_{j-1}||\mathsf{Auth}_{j-1}) \oplus B_j), & \text{if } \lfloor i/2^j \rfloor \equiv 0 \mod 2 \\ \mathrm{H}((\mathsf{Auth}_{j-1}||P_{j-1}) \oplus B_j), & \text{if } \lfloor i/2^j \rfloor \equiv 1 \mod 2 \end{cases}$$

for $0 \leq j \leq h$. If P_h is equal to the root of the XMSS tree given in the public key, the signature is accepted. Otherwise, it is rejected.

3 XMSS$^+$: On-Card Key Generation

In [22], a hash-based signature scheme similar to XMSS is implemented on smart cards. But they did not implement on-card key generation, because of the heavy computations required. In this section we introduce XMSS$^+$, which allows for fast on-card key generation. The techniques used are based on the tree chaining technique introduced in [9] and distributed signature generation from [7]. The basic idea is the following. To obtain an instance of XMSS$^+$ that can be used to make 2^h signatures, we use two levels of XMSS key pairs with height $h/2$ instead of one key pair with height h: One key pair on the upper level ($\mathcal{U}$) of height $h/2$ is used to sign the roots of $2^{h/2}$ key pairs on the lower level ($\mathcal{L}s$) of height $h/2$. The root of $\mathcal{U}$ becomes the public key and the $\mathcal{L}s$ are used to sign the messages. During key generation, $\mathcal{U}$ and the first $\mathcal{L}$ are generated. The generation of the remaining $\mathcal{L}s$ is distributed among signature generations. As a result, the time to generate a key pair, that can be used to sign 2^h messages, goes down from $\mathcal{O}(2^h)$ to $\mathcal{O}(2^{h/2})$.

A signature always contains the current index, the signature of the message using the current $\mathcal{L}$, and the signature of the root of $\mathcal{L}$ under $\mathcal{U}$. To decrease the worst case signing time, the authors of [7] propose to equally distribute the costs for signing the roots of the $\mathcal{L}s$ among the message signatures. For XMSS$^+$ we propose a new approach to distribute these costs. We use the observation that the BDS algorithm does not always use all updates it receives. These unused updates can be used to compute the signatures of the roots from the $\mathcal{L}s$. Thereby we reduce the worst case signing time, again. We use the same bit masks and the same X value for all trees. Thereby the public key size is reduced, as it contains less bit masks. To generate the secret keys, we select a random initial state for FsGen for each key pair, just in time. Now we describe the key generation, signature generation and signature verification algorithms in detail.

Key generation. The XMSS$^+$ key generation algorithm takes as inputs the security parameter n, the message length m, the hash function H, the function family $\mathcal{F}$, and the overall height h, h is even. We set the internal tree height $h' = h/2$. In contrast to the last section, it takes two Winternitz parameters w_u, w_l and two BDS parameters k_u, k_l such that $h' - k_i$ is even for $i \in \{l, u\}$ and $(h' - k_u)/2 + 1 \leq 2^{h' - k_l + 1}$. As for XMSS, the bitmasks and the X are chosen uniformly at random, but this time $h' + \max\{\log \ell_u, \log \ell_l\}$ bitmasks are chosen. Both, the bitmasks and the X are used for both levels. Then the two XMSS key pairs $\mathcal{L}$ and $\mathcal{U}$ are generated. This is done as described in the last section. For $\mathcal{L}$, w_l, k_l, and the message length m are used. For $\mathcal{U}$, w_u and k_u are used. The message length for $\mathcal{U}$ is n, because this is the size of the root nodes of the $\mathcal{L}s$. Next, the root of $\mathcal{L}$ is signed using the first W-OTS keypair of $\mathcal{U}$. Then, a FsGen state for the next $\mathcal{L}$ is chosen uniformly at random, and a new TreeHash stack Stack_{next} is initialized.

The XMSS$^+$ secret key SK consists of the two FsGen states S_l and S_u and the BDS states $\mathsf{State}_{\mathrm{BDS},l}$ and $\mathsf{State}_{\mathrm{BDS},u}$ for $\mathcal{U}$ and $\mathcal{L}$ and the signature on the root of $\mathcal{L}$. Additionally, it contains a FsGen state S_n, a TreeHash stack Stack_{next} and a BDS state $\mathsf{State}_{\mathrm{BDS},n}$ for the next $\mathcal{L}$. The public key PK consists of the $h' + \max\{\log \ell_1, \log \ell_2\}$ bitmasks, the value X and the root of $\mathcal{U}$.

Signature generation. The signature generation algorithm takes as input a message M, the secret key SK, and the index i. First, M is signed. This is done as described in the last section, using S_l and $\mathsf{State}_{\mathrm{BDS},l}$ as secret key for $\mathcal{L}$ and $i \mod 2^{h'}$ as index. During this signature generation, BDS receives $(h' - k_l)/2$ updates. If not all of these updates are used to update $\mathsf{State}_{\mathrm{BDS},l}$, the remaining updates are used to update $\mathsf{State}_{\mathrm{BDS},u}$. Then one leaf of the next lower tree is computed and used as input for TreeHash to update Stack_{next}. The signature $\Sigma = (\sigma_u, \mathsf{Auth}_u, \sigma_l, \mathsf{Auth}_l, i)$ contains the one-time signatures from $\mathcal{U}$ and $\mathcal{L}$ and the two authentication paths, as well as the index i.

If $i \mod 2^{h'} = 2^{h'} - 1$ the last W-OTS key pair of the current $\mathcal{L}$ was used. In this case, Stack_{next} now contains the root of the next $\mathcal{L}$. Now, $\mathcal{U}$ is used to sign this root. The key pair consists of S_u and $\mathsf{State}_{\mathrm{BDS},u}$. The used index is $\lceil i/2^{h'} \rceil$. In contrast to the signing algorithm from the last section, BDS receives no updates at this time. The updates needed to compute the next authentication path are received during the next $2^{h'}$ message signatures. In SK $\mathsf{State}_{\mathrm{BDS},l}$, S_l, and the signature of the root of the $\mathcal{L}$ are replaced by $\mathsf{State}_{\mathrm{BDS},n}$, S_n and the new computed signature, respectively. Afterwards, the data structures for the next $\mathcal{L}$ are initialized and used to replace the ones in SK.

Signature verification. The signature verification algorithm takes as input a signature $\Sigma = (\sigma_u, \mathsf{Auth}_u, \sigma_l, \mathsf{Auth}_l, i)$, the message M and the public key PK. To verify the signature, M and σ_l are used to construct the corresponding W-OTS public key, and then the corresponding leaf node. This leaf node, Auth_l and the index $j = i \mod 2^{h'}$ are used to compute the root of $\mathcal{L}$. This root in turn, is used together with σ_u to compute the W-OTS public key and the corresponding leaf node of $\mathcal{U}$. This leaf node, Auth_u and the index $j = \lfloor i/2^{h'} \rfloor$ are used to compute a root for $\mathcal{U}$. The root computations are done as described in the last section. If the resulting root equals the root node included in the public key, the signature is accepted and rejected otherwise.

3.1 Analysis

In the following we provide an analysis of XMSS$^+$. We show that the distributed authentication path computation works and revisit the security of the scheme. We start with key and signature sizes and the runtimes of the algorithms. A theoretical comparison with XMSS will be included in the full version of this paper.

Sizes and Runtimes. First we look at the sizes. The signature size grows by the size of one W-OTS signature and is $(h+\ell_u+\ell_l)n$ bits. The public key size slightly decreases, as the number of bitmasks decreases and is $(h+2\max\{\log\ell_u,\log\ell_l\}+2)n$ bits. The secret key stays about the same size, depending on the parameter choices, and is at most $(7.5h-7k_l-5k_u+2^{k_l}+2^{k_u}+\ell_u)n$ bits. For the runtimes we only look at the worst case times and get the following. The key generation time is reduced to $2^{h/2}(\ell_u+\ell_l+2)t_{\mathrm{H}}+2^{h/2}(4+\ell_u(w_u+1)+\ell_l(w_l+1))t_{\mathrm{F}}$, where t_{H} and t_{F} denote the runtimes of one evaluation of H and F, respectively. The worst case signing time also decreases because the trees are smaller and requires less than $\max_{i\in\{l,u\}}\{(((h'-k_l+2)/2)\cdot(h'-k_i+\ell_i)+h')t_{\mathrm{H}}+(((h'-k_l+4)/2)\cdot(\ell_i(w_i+1))+h'-k_l)t_{\mathrm{F}}\}$ (Recall that $h'=h/2$). Signature verification increases by the costs of verifying one W-OTS signature and computing the corresponding leaf. It requires $(\ell_u+\ell_l+h)t_{\mathrm{H}}+(\ell_u w_u+\ell_l w_l)t_{\mathrm{F}}$.

Correctness. In the following we show, that the unused updates from $\mathcal{L}$ suffice to compute the authentication paths and to sign the next root. For the computation of the ith authentication path Auth_i in $\mathcal{U}$ and the signature on the $(i+1)$th root, all unused updates from the $(i-1)$th $\mathcal{L}$ can be used. The signature algorithm spends $(h'-k_l)/2$ updates per signature. Hence, the BDS algorithm receives $(h'-k_l)2^{h'-1}$ updates while the $(i-1)$th $\mathcal{L}$ is used. For all authentication paths of $\mathcal{L}$, the BDS algorithm has to compute all right nodes of the tree, that are on a height $< h'-k_l$, besides the two first right nodes on every height as these nodes are already stored during initialization. The number of required updates for $2\le k_l\le h'$ is

$$\sum_{i=0}^{h'-k_l-1}(2^{h'-i-1}-2)2^i=(h'-k_l)2^{h'-1}-2^{h'-k_l+1}$$

so there are $(h'-k_l)2^{h'-1}-(h'-k_l)2^{h'-1}+2^{h'-k_l+1}=2^{h'-k_l+1}$ unused updates. As $(h'-k_u)/2+1\le 2^{h'-k_l+1}$, the BDS algorithm for the $\mathcal{U}$ receives all $(h'-k_u)/2$ updates to compute Auth_i before it is needed and one update is left for the signature on the next root. Doing the same computation for $k_l=0$ there are even more $(3\cdot 2^{h'-1})$ unused updates. For $k_l=h'$, it follows from $(h'-k_u)/2+1\le 2^{h'-k_l+1}$ that $k_u=h'$ and therefore all nodes of both trees are stored.

Security. In [6], an exact proof is given which shows that XMSS is forward secure, if $\mathcal{F}$ is a pseudorandom function family and $\mathcal{H}$ a second preimage resistant hash function family. The tree chaining technique corresponds to the product composition from [19]. In [19] the authors give an exact proof for the forward security of the product composition if the underlying signature schemes are forward secure. It is straight forward to combine both security proofs to obtain an exact proof for the forward security of XMSS$^+$.

4 Implementation

In this section we present our smart card implementation. First we give a description of our implementation. Then we present our results and give a comparison with XMSS, RSA and ECDSA. At the end of the section we discuss an issue regarding the non-volatile memory (NVM).

Implementation Details. For the implementation we use an Infineon SLE78 CFLX4000PM offering 8 KB RAM and 404 KB NVM. Its core consists of a 16-bit CPU running at 33 MHz. Besides other peripherals, it provides a True Random Number Generator (TRNG), a symmetric and an asymmetric crypto co-processor. We use the hardware accelerated AES implementation of the card to implement the function families $\mathcal{F}$ and $\mathcal{H}$. As proposed in [6], we use plain AES for $\mathcal{F}$. To implement $\mathcal{H}$ we build a compression function using the Matyas-Meyer-Oseas construction [20] and iterate it using the Merkle-Darmgard construction [12, 21]. As the input size of $\mathcal{H}$ is fixed, we do not require M-D strengthening. Figure 2 shows the whole construction. As shown there, the construction requires two AES evaluations per evaluation of $\mathrm{H}_K \in \mathcal{H}$. All random inputs of the scheme are generated using the TRNG. Besides XMSS$^+$, we also implemented XMSS for comparison.

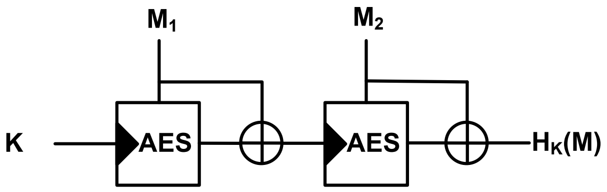

Fig. 2. Construction of $\mathcal{H}$ using AES with the Matyas-Meyer-Oseas construction in M-D Mode

Results. Tables 1 and 2 show the runtimes of our implementation with different parameter sets. We use the same k and w for both trees. The last column shows the security level for the given parameter sets. Following the updated heuristic of Lenstra and Verheul [18] the configurations with a security level of 81 (85, 86) bits are secure until the year 2019 (2025, 2026). In Appendix A we explain how the security level is computed. Please note that these numbers represent a lower bound on the provable security level. A successful attack would still require an adversary to either find a second preimage in a 128 bit hash function or to launch a successful key retrieval attack on AES 128. This would result in 128 bit security for all parameter sets. In Table 1, the signature time is the worst case time over all signatures of one key pair. The secret key size in the table differs from the values we would obtain using the theoretical formulas from the last section. This is because it includes all data that has to be stored on the card to generate signatures, including the bitmasks and X.

Table 1. Results for XMSS and XMSS$^+$ for message length $m = 256$ on an Infineon SLE78. We use the same k and w for both trees. b denotes the security level in bits. The signature times are worst case times.

				Timings (ms)			Sizes (byte)			
Scheme	h	k	w	KeyGen	Sign	Verify	Secret key	Public key	Signature	b
XMSS$^+$	16	2	4	5,600	106	25	3,760	544	3,476	85
XMSS$^+$	16	2	8	5,800	105	21	3,376	512	2,436	81
XMSS$^+$	16	2	16	6,700	118	22	3,200	512	1,892	71
XMSS$^+$	16	2	32	10,500	173	28	3,056	480	1,588	54
XMSS$^+$	20	4	4	22,200	106	25	4,303	608	3,540	81
XMSS$^+$	20	4	8	22,800	105	21	3,920	576	2,500	77
XMSS$^+$	20	4	16	28,300	124	22	3,744	576	1,956	67
XMSS$^+$	20	4	32	41,500	176	28	3,600	544	1,652	50
XMSS	10	4	4	14,600	86	22	1,680	608	2,292	92
XMSS	10	4	16	18,800	100	17	1,648	576	1,236	78
XMSS	16	4	4	925,400	134	23	2,448	800	2,388	86
XMSS	16	4	16	1,199,100	159	18	2,416	768	1,332	72

We used parameter sets with two heights. A key pair with $h = 16$ allows to generate more than $65,000$, one with $h = 20$ to generate more than one million signatures. Assuming a validity period of one year, this corresponds to seven signatures per day and two signatures per minute, respectively. The runtimes show, that XMSS$^+$ key generation can be done on the smart card in practical time. For all but one used parameter set, the key generation time is below 30 seconds. The times for signature generation and verification are all below 200 ms and 30 ms, respectively. The size of the secret key is around four kilo byte and signatures are around two kilo byte, while the public keys are around 500 bytes. Increasing the tree height for XMSS almost doubles key generation time. For XMSS$^+$ the key generation time is almost doubled if one increases the height by two, as this means that the height of each internal tree is increased by one.

The results show that we can reduce the signature size by increasing the Winternitz parameter w. The behavior of the implementation reflects the theory. The factor for the reduction of the W-OTS signature size is only logarithmic in w. The increase of the runtime is negligible for small w. This can be explained by the following. While the length of the single function chains increases, the number of chains decreases. For $w > 16$ the increase of the runtime becomes almost linear. So from this point, $w = 16$ seems to be a good choice. On the other hand, the provable security level also decreases almost linearly in w. While this only reflects a provable lower bound on the security of the scheme, it is still another reason to keep w small.

Table 2. Results for XMSS$^+$ for message length $m = 256$ on an Infineon SLE78 for different values of k. We use the same k and w for both trees. The table shows the worst case signing times, as well as the average case times.

				Timings (ms)			Size (byte)
Scheme	h	k	w	KeyGen	Sign (w.c.)	Sign (avg.c.)	Secret key
XMSS$^+$	16	0	16	6,700	133	96	3,312
XMSS$^+$	16	2	16	6,700	118	96	3,200
XMSS$^+$	16	4	16	6,700	97	83	3,232
XMSS$^+$	16	6	16	7,000	95	67	4,352
XMSS$^+$	16	8	16	8,000	94	53	10,112

Table 2 shows two things. On the one hand, it is possible to decrease the average case signing time spending more storage for the secret key state, by increasing k. This is what one assumes given the theory. On the other hand, the worst case signing time can only be reduced up to a certain limit. For the given parameters this limit is 94ms, the worst case signing time, when both trees are completely stored. These 94ms are mainly caused by the write operations, when one key pair on the lower level is finished. While all the computations are done in previous rounds, the data structures for the next lower level key pair have to be copied to the data structure for the current lower level key pair. Further the new data structures for the next lower level key pair must be initialized. Choosing $k = 4$ seems to be the most reasonable choice for $h = 16$.

Comparison. The last rows of Table 1 show the results for classical XMSS. The results show that XMSS key generation can be done on the smart card, but is impractical as it already takes more than 15 minutes for $h = 16$. Increasing the height by one almost doubles the runtime of key generation. Generating a key with XMSS$^+$ is already for $h = 16$ almost 200 times faster than with XMSS. While XMSS$^+$ signature generation is slightly faster for comparable parameters, verification is faster for XMSS. The faster key generation is paid by slightly bigger secret keys and signatures, while the XMSS$^+$ public keys are smaller, because of the reused bitmasks.

Now we compare XMSS$^+$ with RSA 2048 and ECDSA 256 on the same smart card. The key generation performance of XMSS$^+$ is similar to RSA 2048, which needs on average 11 seconds, but slower than ECDSA 256 (95ms). Signature generation is comparable to RSA 2048 (190ms) and ECDSA 256 (100ms). Only verification takes slightly longer than with RSA 2048 (7ms), but it is faster than with ECDSA 256 (58ms). The security level of RSA 2048 and ECDSA 256 is 95 and 128 bits, respectively. In contrast to the security level shown in Table 1, these numbers are not based on a security proof, but on the best known attacks. As mentioned above, the security level of XMSS$^+$ is 128 bit, when we only assume the best known attacks.

NVM. The changing key presents a challenge for the implementation of XMSS$^+$ and XMSS on smart cards. NVM is organized in sectors and pages. Due to physical limitations only complete pages can be written (erased and reprogrammed) at once. Furthermore they wear out and cannot be programmed anymore after a certain number of write cycles, depending on the technology (about $500,000$ in our case). However, as write operations are distributed over all 33 physical pages of a sector, the complete available cycles are around 16.5 million per sector.

Generating a key takes only a few hundred write cycles, but its state has to be updated after each signature step. Overall, one million available signatures require one million write cycles for the modification of the state. Using careful memory management, layout and optimization, we managed to keep the number of write cycles below five million for a key pair with $h = 20$, which is far below the 16.5 million available per sector. This includes key generation and all 2^{20} signatures. It should be noted, that this affects only one NVM sector of the card. To use multiple keys, they can be placed in different sectors in order to preserve NVM quality.

5 Conclusion

We presented the first smart card implementation of a forward secure signature scheme. The results presented in Section 4 show that the implementation is practical and that key generation can be done on the card in less than a minute. This is in contrast to previous implementations of similar schemes, that did not achieve on-card key generation. To achieve this, we introduced XMSS$^+$, an improved version of XMSS. Besides the improved key generation, the worst case signing time is also reduced. While the presented improvement is necessary for an implementation on smart cards, it might also show to be useful for implementations on other hardware (At least in cases, where key generation time or worst case signing time are critical).

Given the results of the last section, we propose the parameter set $h = 16$, $w = 16$ and $k = 4$. These parameters seem to lead the optimal performance as long as $65,000$ signatures per key pair are enough. The provable lower bound on the security level of 71 bits is too low from a theoretical point of view. But if we compute the security level according to the best known attacks - as it is common practice - we get a security level of 128 bit. This leads to interesting directions for future work. One would be to either tighten the security proofs or find better reductions from different security assumptions. Another one would be to implement XMSS$^+$ with co-processors for block ciphers with a bigger block size than AES. Alternatively, it would be possible to use hash functions with a digest length of more than 128 bit, using the constructions from [6] to construct the PRF.

One topic we did not address in this work is the side channel resistance. But the forward security property already protects against the most common attack vector for side channel attacks. If a user looses her smart card and revokes her key pair, an attacker can not gain any advantage of a successful side channel attack. The secret key the adversary learns is revoked from this time on and it is not possible to learn the keys of prior time periods. Nevertheless, as there exist other attack vectors, it would be interesting to analyze the side channel resistance of our implementation.

References

1. Abdalla, M., Miner, S.K., Namprempre, C.: Forward-Secure Threshold Signature Schemes. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 441–456. Springer, Heidelberg (2001)
2. Abdalla, M., Reyzin, L.: A New Forward-Secure Digital Signature Scheme. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 116–129. Springer, Heidelberg (2000)
3. Anderson, R.: Two remarks on public key cryptology. Relevant Material Presented by the author in an Invited Lecture at the 4th ACM Conference on Computer and Communications Security, CCS, pp. 1–4. Citeseer (1997) (manuscript)
4. Bellare, M., Miner, S.K.: A Forward-Secure Digital Signature Scheme. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (1999)
5. Buchmann, J., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the Security of the Winternitz One-Time Signature Scheme. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 363–378. Springer, Heidelberg (2011)
6. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011)
7. Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle Signatures with Virtually Unlimited Signature Capacity. In: Katz, J., Yung, M. (eds.) ACNS 2007. LNCS, vol. 4521, pp. 31–45. Springer, Heidelberg (2007)
8. Buchmann, J., Dahmen, E., Schneider, M.: Merkle Tree Traversal Revisited. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 63–78. Springer, Heidelberg (2008)
9. Buchmann, J., García, L.C.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – An Improved Merkle Signature Scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 349–363. Springer, Heidelberg (2006)
10. Camenisch, J., Koprowski, M.: Fine-grained forward-secure signature schemes without random oracles. Discrete Applied Mathematics 154(2), 175–188 (2006); Coding and Cryptography
11. Cronin, E., Jamin, S., Malkin, T., McDaniel, P.: On the performance, feasibility, and use of forward-secure signatures. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, pp. 131–144. ACM, New York (2003)
12. Damgård, I.B.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)

13. ETSI. XML advanced electronic signatures (XAdES). Standard TS 101 903, European Telecommunications Standards Institute (December 2010)
14. ETSI. CMS advanced electronic signatures (CAdES). Standard TS 101 733, European Telecommunications Standards Institute (March 2012)
15. Itkis, G., Reyzin, L.: Forward-Secure Signatures with Optimal Signing and Verifying. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 332–354. Springer, Heidelberg (2001)
16. Kozlov, A., Reyzin, L.: Forward-Secure Signatures with Fast Key Update. In: Cimato, S., Galdi, C., Persiano, G. (eds.) SCN 2002. LNCS, vol. 2576, pp. 241–256. Springer, Heidelberg (2003)
17. Krawczyk, H.: Simple forward-secure signatures from any signature scheme. In: Proceedings of the 7th ACM Conference on Computer and Communications Security, CCS 2000, pp. 108–115. ACM, New York (2000)
18. Lenstra, A.K.: Key lengths. Contribution to the Handbook of Information Security (2004)
19. Malkin, T., Micciancio, D., Miner, S.K.: Efficient Generic Forward-Secure Signatures with an Unbounded Number Of Time Periods. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 400–417. Springer, Heidelberg (2002)
20. Matyas, S., Meyer, C., Oseas, J.: Generating strong one-way functions with cryptographic algorithms. IBM Technical Disclosure Bulletin 27, 5658–5659 (1985)
21. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
22. Rohde, S., Eisenbarth, T., Dahmen, E., Buchmann, J., Paar, C.: Fast Hash-Based Signatures on Constrained Devices. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 104–117. Springer, Heidelberg (2008)
23. Song, D.X.: Practical forward secure group signature schemes. In: Proceedings of the 8th ACM Conference on Computer and Communications Security, CCS 2001, pp. 225–234. ACM, New York (2001)

A Security Level

We compute the security level in the sense of [18]. This allows a comparison of the security of XMSS$^+$ with the security of a symmetric primitive like a block cipher for given security parameters. Following [18], we say that XMSS$^+$ has security level b if a successful attack on the scheme can be expected to require approximately 2^{b-1} evaluations of functions from F_n and $\mathcal{H}_n$. Following the reasoning in [18], we only take into account generic attacks on $\mathcal{H}_n$ and F_n. A lower bound for the security level of XMSS was computed in [6]. For XMSS$^+$, we combined the exact security proofs from [6] and [19]. Following the computation in [6], we can lower bound the security level b by

$$b \geq \max\left\{n - h/2 - 4 - w_u - 2log(\ell_u w_u), n - h - 4 - w_l - 2log(\ell_l w_l)\right\}$$

for the used parameter sets.