Fast Hash-Based Signatures on Constrained Devices

Sebastian Rohde[1], Thomas Eisenbarth[1], Erik Dahmen[2], Johannes Buchmann[2], and Christof Paar[1]

[1] Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany
{rohde,eisenbarth,cpaar}@crypto.rub.de

[2] Technische Universität Darmstadt
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{dahmen,buchmann}@cdc.informatik.tu-darmstadt.de

Abstract. Digital signatures are one of the most important applications of microprocessor smart cards. The most widely used algorithms for digital signatures, RSA and ECDSA, depend on finite field engines. On 8-bit microprocessors these engines either require costly coprocessors, or the implementations become very large and very slow. Hence the need for better methods is highly visible. One alternative to RSA and ECDSA is the Merkle signature scheme which provides digital signatures using hash functions only, without relying on any number theoretic assumptions. In this paper, we present an implementation of the Merkle signature scheme on an 8-bit smart card microprocessor. Our results show that the Merkle signature scheme provides comparable timings compared to state of the art implementations of RSA and ECDSA, while maintaining a smaller code size.

Keywords: *Embedded security, hash based cryptography, Merkle signature scheme, digital signatures.*

1 Motivation

Smart cards are used in many areas of every day life. Application areas include payment systems, electronic health cards and SIM cards for mobile phones. With the advent of contactless smart cards, new and important fields of application have recently emerged, like the electronic passport, which is now deployed in many countries, especially in Europe and the US. Other countries, like Belgium, also issue electronic ID cards to their citizens [11].

The most important application of smart cards is secure identification and authentication. Many of the above mentioned applications have a need for strong security. All these requirements are met by digital signatures. Digital signatures provide authenticity, integrity and support for non-repudiation of data and are often used in identification and authentication protocols for smart cards.

Since smart cards are usually provided in high quantities, there is also a need to keep costs as low as possible. This is one of the reasons why most of the microprocessor cards in use are still equipped with small and cheap 8-bit CPUs. These small 8-bit microprocessors are constrained in program memory (flash or ROM), RAM, clock speed, register width, and arithmetic capabilities.

Common signature schemes such as RSA and ECDSA require operations in a finite field for the signature generation and verification. For efficient implementations in smart cards, costly coprocessors that implement the field arithmetic are required. In 1979 Merkle proposed a signature scheme that requires only hash function evaluations for the signature generation and verification [20]. Since software implementations of hash functions are much more efficient than software implementations of finite field arithmetic, the Merkle signature scheme (MSS) is a good candidate for implementations on small microprocessors without cryptographic coprocessors. Another benefit of the MSS is the fact that its security relies only on the cryptographic properties of the used hash function and not on additional number theoretic assumptions. If the hash function used for the MSS is found insecure, it can be replaced by a secure one to obtain a new and secure instance of the MSS.

Our Contribution. In this paper we present an implementation of the Merkle signature scheme for 8-bit Atmel AVR microcontrollers, e.g. smart card processors from the AT90SCxxx family. Our implementation is highly scalable and can be configured to provide an ideal tradeoff between security, execution times, and memory requirements for the specific use case. We will show that our implementation of the MSS performs excellently when compared to RSA and ECDSA. Our implementation has a smaller code size and faster verification times. The signature generation is faster than RSA and comparable to ECDSA. Further performance improvements are reached by utilizing a symmetric crypto engine such as an AES hardware acceleration.

For the underlying hash functions we use constructions that are based on the AES block cipher. Such hash functions have two advantages compared to dedicated hash functions: (1) they have a small block size which is more suitable for the MSS and (2) they are more efficient in size and speed.

Related Work. Gura *et al.* showed the feasibility of public key cryptography on constrained 8-bit microcontrollers. Their implementation of RSA-1024 and RSA-2048 showed that digital signatures are feasible on 8-bit platforms even without expensive crypto-coprocessors. Further research regarding digital signatures on constrained 8-bit devices has been performed in the field of wireless sensor networks. Liu and Ning published a full ECC engine called TinyECC which also does not require a coprocessor. They implemented the 160-bit elliptic curve *secp160r1*. Winternitz one-time signatures have also been proposed to be used in wireless sensor networks for signing short messages (< 80 bit) [18]. The proposed solution, however, uses a public key management that is not applicable to smart cards. Others [24, 7] show possible use cases for MSS on constrained devices without making any suggestions regarding the implementation.

Organization. The paper is organized as follows: Section 2 gives an overview of the MSS and the used hash functions. Section 3 explains the target platform and details about the implementation. Section 4 presents performance results and a comparison. Section 5 elaborates a possible performance gain when using an AES hardware acceleration. Section 6 states our conclusion.

2 Preliminaries

In this section we describe the details of the variant of the Merkle signature scheme [20] we use for our implementation. In summary we use the Winternitz one-time signature scheme (W-OTS) [9] to sign the data, the ideas for efficient one-time signature key generation of [4] and the algorithm from [6] for the computation of the authentication paths. We use two different hash functions based on the AES block cipher, both with 128-bit block length. We use a 256-bit hash function for the initial hashing (digest creation) of the data to be signed and a 128-bit hash function for the one-time signature scheme and the Merkle tree. Details on the construction of these hash functions are described in Section 2.2.

2.1 The Merkle Signature Scheme

We now describe the three algorithms for the key generation, signature generation, and verification. In the following, let $F : \{0,1\}^* \rightarrow \{0,1\}^{128}$ and $G : \{0,1\}^* \rightarrow \{0,1\}^{256}$ be cryptographic hash functions.

Key Generation. The first step of the key generation is to decide how many signatures should be generated with this key pair. We choose the parameter $H \geq 2$ to be able to generate 2^H signatures. The next step is to generate 2^H W-OTS key pairs. For the W-OTS key generation, we apply the approach of [4] and use the following forward secure pseudo random number generator.

$$\text{PRNG} : \{0,1\}^{128} \rightarrow \{0,1\}^{128} \times \{0,1\}^{128}, \textsc{Seed}_{\text{in}} \mapsto (\textsc{Seed}_{\text{out}}, \textsc{Rand}).$$

As suggested in [5], we use the hash based PRNG proposed in [12], i.e.

$$\textsc{Rand} \leftarrow F(\textsc{Seed}_{\text{in}}), \textsc{Seed}_{\text{out}} \leftarrow (1 + \textsc{Seed}_{\text{in}} + \textsc{Rand}) \bmod 2^{128}.$$

The MSS private key is an 128-bit seed Seed chosen uniform at random. This seed is fed to the PRNG to compute the initial seed $\textsc{Seed}_{\text{W-OTS}}$ that we use to generate first W-OTS signature key:

$$(\textsc{Seed}, \textsc{Seed}_{\text{W-OTS}}) = \text{PRNG}(\textsc{Seed}). \tag{1}$$

Doing so, Seed is updated and can be used to compute the initial seeds for upcoming W-OTS signature keys. Depending on the Winternitz parameter w, the W-OTS signature key consists of $t = t_1 + t_2$ 128-bit strings, where

$$t_1 = \left\lceil \frac{256}{w} \right\rceil, \quad t_2 = \left\lfloor \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rfloor.$$

The W-OTS signature key is the sequence $X = (x_1, \ldots, x_t)$, that consists of t bit strings each of length 128-bit. It is computed using the PRNG as

$$(\text{SEED}_{\text{W-OTS}}, x_i) = \text{PRNG}(\text{SEED}_{\text{W-OTS}}) \tag{2}$$

for $i = 1, \ldots, t$. The W-OTS verification key is $Y = F(y_1 \parallel \ldots \parallel y_t)$, where $y_i = F^{2^w-1}(x_i)$, i.e. the hash function F is applied $2^w - 1$ times to x_i for $i = 1, \ldots, t$.

The 2^H W-OTS verification keys are the leaves of the Merkle tree. The inner nodes are computed using the following construction rule: a parent node is the hash of the concatenation of its left and right children, i.e.

$$\text{Node}_{\text{parent}} = F(\text{Node}_{\text{left child}} \parallel \text{Node}_{\text{right child}}).$$

By applying this rule iteratively the root of the Merkle tree, which is also the MSS public key, is obtained.

Signature Generation. To sign some data, the first step is to compute its 256-bit digest: $d = G(\text{data})$. The W-OTS signature keys are used sequentially. We describe the generation of the sth signature, $s \in \{0, \ldots, 2^H - 1\}$. The sth W-OTS signature key is computed from the seed SEED as described in Equations (1) and (2). We always update the seed in the private key and therefore one invocation of the PRNG suffices to obtain the initial seed $\text{SEED}_{\text{W-OTS}}$ to compute the sth W-OTS signature key. The Winternitz signature of d is then computed as follows: (1) split the binary representation d into t_1 blocks $b_1, \ldots, b_{t_1}$ each of length w. (2) Consider b_i as the integer encoded by this block in binary and compute $c = \sum_{i=1}^{t_1}(2^w - b_i)$. (3) Split the binary representation c into t_2 blocks $b_{t_1+1}, \ldots, b_t$ each of length w. If the bit-length of c or d is no multiple of w we pad with zeros to the left. The Winternitz signature of d is then given as $\sigma_{\text{W-OTS}}(d) = (\sigma_1, \ldots, \sigma_t)$, where $\sigma_i = F^{b_i}(x_i)$, for $i = 1, \ldots, t$. The sth MSS signature of d is given as

$$\sigma_s(d) = \big(s, \sigma_{\text{W-OTS}}(d), (a_0, \ldots, a_{H-1})\big).$$

The sequence $(a_0, \ldots, a_{H-1})$ is the authentication path for the sth leaf, i.e. the sth W-OTS verification key. It is defined as the siblings of all nodes on the path from the sth leaf to the root of the Merkle tree, see Figure 1. For the computation of authentication paths we use the BDS algorithm from [6]. This algorithm is constructed such that the authentication path for the currently used leaf is already available and the upcoming authentication paths are prepared after the MSS signature is computed. The BDS algorithm uses a parameter $K \geq 2$ which decides how many nodes close to the root are stored during the initialization to reduce the computational cost. The initialization of this algorithm, that requires certain tree nodes to be stored, is done during the MSS key generation.

Signature Verification. The first step of the signature verification is again to compute the digest of the data that was signed: $d = G(\text{data})$. Then d and its

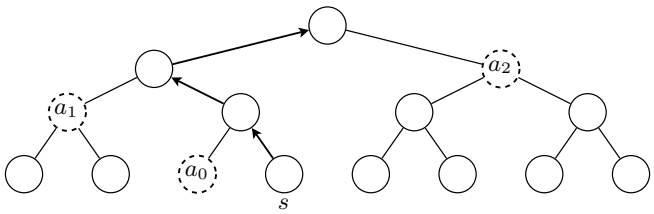

Fig. 1. Example of the Merkle signature scheme for $H = 3, s = 3$. Dashed nodes denote the authentication path for the sth leaf. The arrows indicate the path from the sth leaf to the root.

Winternitz signature are used to compute the sth leaf as follows: Repeat steps (1)-(3) of the Winternitz signature generation to obtain $b_1, \ldots, b_t$. The sth leaf φ is computed as

$$\varphi = F\big(F^{2^w-1-b_1}(\sigma_1) \parallel \ldots \parallel F^{2^w-1-b_t}(\sigma_t)\big)$$

Then the path from the sth leaf to the root and the root itself is recomputed using the authentication path and the index s:

$$\varphi = \begin{cases} F(\varphi \parallel a_h) \text{ , if } s/2^h \equiv 0 \bmod 2 \\ F(a_h \parallel \varphi) \text{ , if } s/2^h \equiv 1 \bmod 2 \end{cases}$$

for $h = 0, \ldots, H-1$. If the computed root matches the signers public key, the signature is valid.

Time and Memory Requirements. We now estimate the number of evaluations of F required for the key generation, signature generation and verification. We also estimate the storage requirements of the public and private key and the signatures.

The MSS key generation requires the computation of 2^H leaves or W-OTS key pairs and $2^H - 1$ evaluations of F to compute the root. The computation of one leaf costs $t(2^w - 1) + 1$ evaluations of F and $t + 1$ calls to the PRNG. Using that one call to the PRNG costs as much as one evaluation of F, the key generation in total requires $2^H(t2^w + 3) - 1$ evaluations of F. The public key requires 128 bits of memory.

For each signature, the BDS algorithm requires at most $(H - K)/2 + 1$ leaves, $3(H - K - 1)/2 + 1$ evaluations of F and $H - K$ calls to the PRNG to compute upcoming authentication paths. If s is even the BDS algorithm requires $(H - K)/2 + 1$ leaves to be computed. One of these leaves is the sth leaf. Since the Winternitz signature of the data just signed using the sth W-OTS key is an intermediate value during the computation of the sth leaf, the generation of this Winternitz signature needs no additional calculations in this case. If s is odd, the BDS algorithm requires only $(H - K)/2$ leaves to be computed and the Winternitz signature of the data must be computed separately. Since the generation of a Winternitz signature requires less computations than

a leaf, the above cost for the BDS algorithm also represent the total cost for signing. Hence, the total cost for signing in terms of evaluations of F is at most $t2^w(H-K-2)/2+(7H-7K+3)/2$. The BDS algorithm needs to store $3.5H-3K+2^K-2$ nodes of the Merkle tree and $2(H-K)$ seeds which we store as part of the private key. Together with the 128-bit seed used to generate the signature keys, the size of the private key is given as $(5.5H-5K+2^K-1)\cdot 128$ bits. The size of the signature is given as $(t+H)\cdot 128$ bits. $t\cdot 128$ bits for the Winternitz signature and $H\cdot 128$ bits for the authentication path.

The signature verification on average requires $t(2^w-1)/2$ evaluations of F to compute the sth leaf and H evaluations of F to recompute the path to the root and the root itself. The signature generation and verification also require one evaluation of G to compute the initial digest d of the data.

The above formulas show that the Winternitz parameter w provides a time-memory trade-off of the signature size and the key and signature generation times. However, the key and signature generation times of the W-OTS keys are exponential in w, while the signature size decreases only linearly in w. Therefore w should not be chosen too large. Also the output length of the hash functions F and G must be chosen carefully since the size of a Winternitz signature linearly depends on their product. Our choice of 128 and 256 bit yields moderate signature sizes and, as we will explain in the following, high practical security.

Security. The MSS is provably secure against adaptive chosen message attacks, if the used hash function is collision resistant [8]. However, to forge a MSS signature in practice the attacker is required to compute preimages and second-preimages. Therefore the practical security of the MSS currently relies on the preimage and second-preimage resistance of the used hash function [21]. From a practical point of view, the 128-bit hash function F we use for the W-OTS and the Merkle tree provides 128-bit security. Collision resistance is definitely required for the initial hashing of the data to sign. This is why we use the 256-bit hash function G, which provides 128-bit security against collision attacks that exploit the birthday paradox.

2.2 Hash Functions

In this section we present the hash functions that are used in our scheme. Furthermore we show that single and double block length constructions are the better choice when used in conjunction with digital signatures. Relatively short input block lengths and the resulting speed make them better suited for implementations on constrained devices. Public key and private key sizes are proportionally dependent on the hash length of F. As stated earlier, a short value of 128 bit offers adequate security for this scenario while staying within reasonable memory limits. For our scheme, we used the AES algorithm which is specified with a block length of 128 bit. Using AES in a double block length construction leads to a hash length of 256 bit.

Using block ciphers as hash functions in digital signature schemes is also appealing because one primitive can be used for three applications: encryption,

generation of hashes, and digital signatures. In addition block ciphers are much better known and analyzed than dedicated hash functions.

Single Block Length Construction. The single block length hash in our scheme is constructed using a Matyas-Meyer-Oseas (MMO) construction [19]. The MMO construction is recursively defined as $f_{i+1} = E_{f_i}(M_i) \oplus M_i$ with E being the encryption function, M_i as the current message block and f_0 being an initialization vector (See Figure 2). In a hash signature scheme this variant for constructing the hash benefits from the fact that the encryption function always uses the same key (an initialization vector) for the first block.

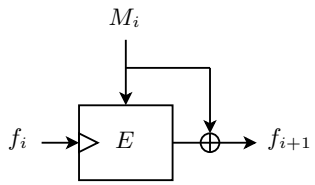

Fig. 2. Single block length compression function due to [19]. The output of the block cipher E is xored with the message block M_i. f_i, f_{i+1}, and M_i are each of bit length 128.

Double Block Length Construction. For applications such as the initial digest generation in a signature scheme, collision resistance is needed and the security of single block length (SBL) constructions is not sufficient. For our implementation, we use MDC-2 double length construction specified in the ISO/IEC 10118-2 standard. The standard envisions the usage of DES, but there is a variant using AES-128 [23], as depicted in Figure 3. This construction takes a block cipher with block length n bit and produces a hash function with $2n$ bit output length. In [22] the authors show, that an adversary needs at least $2^{3n/5}$ oracle queries to find a collision. However, the best practical attacks require 2^n queries.

The double block length construction is only used for initial digest generation. It can easily be replaced by a dedicated hash function resulting in a negligible performance loss but an increased code size.

Comparison to Dedicated Hash Functions. SBL and DBL constructions are much better suited for hash-based signature schemes than dedicated hash functions. A hash function with 512 bit (e.g. whirlpool) length would yield a highly inefficient signature scheme. At the same time it is also interesting to note that dedicated hash functions are optimized for large amounts of data as can be seen by the comparatively large block size (512 bit). With the MSS, the input for the hash function has mainly the same size as the output value. This is

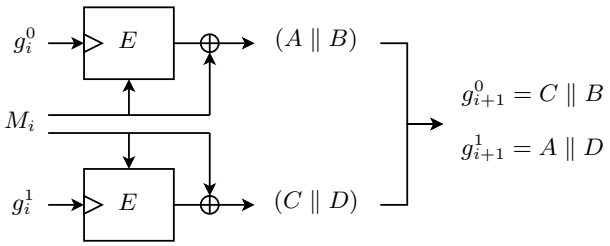

Fig. 3. Double block length compression function due to [23]. The outputs of the block cipher E are xored with the message block M_i and permuted. $g_i^0, g_i^1, g_{i+1}^0 g_{i+1}^1$, and M_i are each of bit length 128.

one of the reasons why dedicated hash functions provide suboptimal performance for appliances in hash based signature schemes.

In addition large block sizes reduce the speed of implementations on the AVR microcontroller platform since the state cannot be held completely in the registers of the processor. In Table 1 we provide a comparison of various hash functions and their performance. The results for the dedicated hash functions are taken from [13] and [15].

Table 1. Performance of hash function implementations on the AVR platform

Hash function	bit length per output	bit length per block	msec per block	cycles per block	cycles per byte
SHA1 [13]	160	512	3.9	63,000	984
SHA1 [15]	160	512	2.6	41,113	642
SHA256 [15]	256	512	3.4	54,196	847
MD5 [13]	128	512	1.5	23,568	368
AES-SBL	128	128	0.3	4,081	255
AES-DBL	256	128	0.5	8,104	507

Concerning the cycles required to hash one block, the block cipher based hash functions provide much better performance. This is due to the large block size of 512 bit used by dedicated hash functions to allow efficient hashing of large amounts of data. This is clarified by the column "cycles per byte" which shows that dedicated hash functions and block cipher based hash functions require a similar time to hash one input byte. However, for the Merkle signature scheme and our choice of parameters most of the time only blocks of length up to 256 bits must be hashed, which requires about 8,000 cycles when using the SBL construction. Hence, SBL constructions are a better choice than dedicated hash functions for the Merkle signature scheme on the target platform.

3 Implementation Details and Target Platform

Target Platform. Our implementation is designed for 8-bit AVR microcontrollers, a popular family of 8-bit RISC microcontrollers. The Atmel AVR processors offer clock speeds of up to 16MHz, a few KBytes of SRAM, up to tens of KBytes of EEPROM and additional flash or mask ROM for program memory. Besides the AVR smart card processors AT90SCxxx [2], AVRs are also available as general purpose microcontrollers with a wide use in many embedded applications. One example is the Atmel ATmega128 microcontroller [3] often used for wireless sensor networks.

The devices of the AVR family have 32 general purpose registers of 8-bit word size. Most of the 130 instructions of the microcontroller are one-cycle. AVR microcontrollers can be programmed in AVR-assembler and in C.

The implementation of this project is designed to be executable on any AVR processor providing 4 KBytes SRAM, about 4 KBytes EEPROM and at least 8 KBytes of program memory. However, for platforms that are even more constrained in available SRAM, our scheme can also be altered to operate on systems with less SRAM. For our implementation the AVR was clocked within specification limits at 16 MHz. We chose to use assembler for performance critical routines such as some cryptographic primitives and C to glue these routines together.

AES Implementation. An efficient AES implementation for the AVR platform is available at [1]. It is licensed under the GPL. We modified this implementation to make it even smaller and faster. In this section we describe our modifications and improvements concerning this AES algorithm.

The RijndaelFurious algorithm is pure assembly code that can be compiled using the Atmel AVR compiler. Some modifications made it compilable using the avr-gcc. In addition the decryption functionality has been removed as it is not necessary for hash function constructions. If an AES decryption is needed, it can be easily added by the cost of a small increase in code size. Furthermore, we contributed our own implementation of the MixColumns function that is better in respect to performance and code size.

The used hash function constructions often apply the initialization vector as the encryption key. For a further speed-up we also implemented an alternative method with a pre-expanded key. This allows to save many key expansions in the process of creating one-block hashes.

Memory Management. The Merkle signature scheme is *key evolving*, which means that after every signing process a modification of the private key is required. The private key needs to be stored in nonfluent memory when the power is lost. Our implementation stores the private key in EEPROM, since the maximum amount of erase/write cycles allowed on the flash memory are usually much more limited than on the EEPROM. Our target platform is specified for at least 100.000 erase/write cycles [2, 3]. Therefore the maximum value for the

height of the Merkle tree supported by our implementation is $H = 16$, which allows $2^{16} = 65.536$ signatures to be generated. We store the whole signature in the SRAM during creation. However, for platforms that are even more constrained concerning SRAM our scheme can easily be altered to support signature generation and verification with much less RAM.

The memory constraints also enforce a very economical way of organizing the data of the private key. Despite of heavy optimizations, some implementation details force the actual size of the private key to be slightly larger than the calculated results from Section 2.1. The main reason is that these formulas count only the number of hash values that must be stored. For example, the stacks used by the BDS algorithm were implemented as arrays of fixed size. In addition to the stack, we need to store the index of the array element that denotes the top node on the stack. Also the size of the signatures is slightly larger than estimated, because the index of the signature must be added as well.

Key Generation. Due to the heavy computations required, the key generation is not done on the microcontroller but on a standard PC. For the generation of test data, we created a PC version of the project that uses mostly the same code base. In contrast to the AVR implementation it uses a different implementation of the AES algorithm and it supports key generation. The key generation is computationally far too costly to run on the microcontroller which is why it has been disabled in the AVR implementation. The speed of the PC version has also been used to verify the correct behavior of our code by iterating through all possible signatures.

Side Channel Resistance. For smart card implementations, resistance against side channel attacks is of high importance. All parts of an implementation handling sensitive data need to be protected against a possible leakage. The W-OTS signature keys X and their seeds SEED are the only critical data. Since the keys X and their seeds are used as one-time keys, the values are being processed very few times, rendering non-template based attacks difficult. However, the analysis of the vulnerability against side channel attacks and, if necessary, the development of efficient countermeasures are an interesting field for future work.

4 Choice of Parameters and Timings

In this section we present the timings of our implementation and the exact memory requirements for the microcontroller. We also compare these values to implementations of state of the art signature schemes on the same microcontroller platform. For the height of the Merkle tree we chose $H = 16$ and $H = 10$ which allows 2^{16} and 2^{10} signatures to be generated with one key pair, respectively. The reason for the choice of $H = 16$ is, that 2^{16} is near to the maximum number of allowed write cycles for the EEPROM of the microcontroller [2, 3]. The values for $H = 10$ were included to clarify the impact of the tree height on the signature generation time. For the Winternitz parameter w and the parameter K for

the BDS algorithm, we tested three combinations $(w, K) = (2, 2), (2, 4), (4, 4)$. The value t, that denotes the number of 128-bit strings in the W-OTS signature key and the one-time signature, is $t = 133$ for $w = 2$ and $t = 67$ for $w = 4$. Table 2 summarizes the results. $s_{\text{pub}}, s_{\text{priv}}, s_{\text{sig}}$, and s_{ROM} denote the memory requirements for the public key, the private key, and the signature as well as the code size in bytes, respectively. t_{verify} and t_{signing} denote the average time in milliseconds required for verification and signature generation, respectively.

Table 2. Timings and memory requirements of our implementation and comparison to state of the art signature schemes on the same platform.

	Memory in bytes				Time in msec	
Scheme	s_{pub}	s_{priv}	s_{sig}	s_{ROM}	t_{verify}	t_{signing}
Our MSS-128 implementation using $H = 16$						
$(w, K) = (2, 2)$	16	1440	2350	6600	85	1230
$(w, K) = (2, 4)$	16	1472	2350	6600	85	1072
$(w, K) = (4, 4)$	16	1472	1330	6600	127	1665
Our MSS-128 implementation using $H = 10$						
$(w, K) = (2, 2)$	16	848	2290	6600	82	756
$(w, K) = (2, 4)$	16	876	2290	6600	82	598
$(w, K) = (4, 4)$	16	876	1234	6600	124	946
RSA-1024 [14]	131	128	128	7400	215	5495
RSA-2048 [14]	259	256	256	10600	970	41630
ECDSA-160 [10]	40	21	40	43200	423	423
ECDSA-160 [17]	40	21	40	17900	1218	1001

Table 2 shows that our implementation features smaller code size and smaller public keys. The above figures already include the code size of the hash function needed for digest generation and the AES engine. Also the signature verification times are faster than those of RSA and ECDSA. The signature generation time of our implementation is much faster than the RSA implementations and comparable to the ECDSA implementations. In case of $H = 10$ our implementation is even faster than the memory efficient ECDSA variant from [17]. The main drawbacks of the MSS are the large memory requirements for the signature and the private key. However, both the private key and the signature easily fit into the EEPROM and the SRAM of the Atmel, respectively.

We finally remark that our implementation provides a practical security of 128 bits and hence offers long term security until the year 2090 [16]. RSA with an 1024-bit modulus offers comparable symmetric security of only 72 bit, i.e. until the year 2006. The security of 2048-bit RSA is at most 95-bit, i.e. until the year 2040. ECDSA using 160-bit elliptic curves offers only 80 bit of security, i.e. until the year 2018. This shows that our implementation is not only very competitive to currently used schemes, but also offers higher practical security [16].

5 Hardware Accelerated AES

The most performance critical part of our MSS implementation is the AES based hash function. Hence, a natural approach to improve the scheme's overall performance is to accelerate the AES implementation. Many recent low-/mid- budget smart card processors offer hardware acceleration for symmetric encryption schemes like the AES. One publicly available platform offering an AES hardware acceleration is the Atmel ATxmega 128A1 processor.

In contrast to the software implementation of the SBL construction using AES that requires 4,081 cycles per block, the hardware accelerated version requires only 1069 cycles per block. However, the AES engine itself needs only 375 cycles on the target platform. Besides the overhead for the hash function construction the remaining cycles are mainly required for the necessary memory management. Table 3 illustrates the performance of the MSS when utilizing AES hardware acceleration. This table shows, that speeding up the hash function by a factor ≈ 3.82 results in a speed up of the Merkle signature scheme by roughly the same factor.

Table 3. Timings and memory requirements of our implementation with and without AES hardware acceleration.

	Memory in bytes				Time in msec	
Scheme	s_{pub}	s_{priv}	s_{sig}	s_{ROM}	t_{verify}	t_{signing}
Our MSS-128 implementation using $H = 16$ / Software AES						
$(w, K) = (2, 2)$	16	1440	2350	6600	85	1230
$(w, K) = (2, 4)$	16	1472	2350	6600	85	1072
$(w, K) = (4, 4)$	16	1472	1330	6600	127	1665
Our MSS-128 implementation using $H = 16$ / Hardware AES						
$(w, K) = (2, 2)$	16	1440	2350	6100	24	362
$(w, K) = (2, 4)$	16	1472	2350	6100	24	317
$(w, K) = (4, 4)$	16	1472	1330	6100	38	504

6 Conclusion

We presented an implementation of the Merkle signature scheme on a low-cost 8-bit microcontroller platform. Our implementation shows that MSS is a competitive signature scheme compared to commonly used signature schemes such as RSA and ECDSA. Our implementation has smaller code size and faster verification times than efficient implementations of RSA and ECDSA. The signature generation times are faster than RSA and comparable to ECDSA.

When employing an available symmetric crypto coprocessor, even further speed up can be reached.

Our implementation gives a positive answer to the question whether highly secure and efficient signature schemes can be implemented on constrained devices.

References

1. Rijndaelfurious implementation, 01 2008. http://point-at-infinity.org/avraes/.
2. Atmel. Overview of secure avr microcontrollers 8-/16-bit risc cpu, 2007. http://www.atmel.com/products/SecureAVR/.
3. Atmel. Specifications of the atmega128 microcontroller, 2007. http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf.
4. J. Buchmann, C. Coronado, E. Dahmen, M. Döring, and E. Klintsevich. CMSS - an improved merkle signature scheme. In R. Barua and T. Lange, editors, *INDOCRYPT*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2006.
5. J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume. Merkle signatures with virtually unlimited signature capacity. In J. Katz and M. Yung, editors, *ACNS*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.
6. J. Buchmann, E. Dahmen, and M. Schneider. Merkle tree traversal revisited. Manuscript, 2008. http://www.cdc.informatik.tu-darmstadt.de/mitarbeiter/dahmen.html.
7. X. Cheng, W. Li, and T. Znati, editors. *Wireless Algorithms, Systems, and Applications, First International Conference, WASA 2006, Xi'an, China, August 15-17, 2006, Proceedings*, volume 4138 of *Lecture Notes in Computer Science*. Springer, 2006.
8. C. Coronado. On the security and the efficiency of the merkle signature scheme. Cryptology ePrint Archive, Report 2005/192, 2005. http://eprint.iacr.org/.
9. C. Dods, N. P. Smart, and M. Stam. Hash based digital signature schemes. In N. P. Smart, editor, *IMA Int. Conf.*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer, 2005.
10. B. Driessen, A. Poschmann, and C. Paar. Comparison of Innovative Signature Algorithms for WSNs. In *Proceedings of the First ACM Conference on Wireless Network Security (to appear)*.
11. ePractice.eu. Belgian electronic ID card officially launched, April 2003. http://www.epractice.eu/document/2139.
12. Digital signature standard (DSS). FIPS PUB 186-2, 2007. Available at http://csrc.nist.gov/publications/fips/.
13. P. Ganesan, R. Venugopalan, P. Peddabachagari, A. Dean, F. Mueller, and M. Sichitiu. Analyzing and modeling encryption overhead for sensor network nodes. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 151–159, New York, NY, USA, 2003. ACM.
14. N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In M. Joye and J.-J. Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2004.
15. D. Labor. Crypto-avr-lib. Available through http://www.das-labor.org/wiki/Crypto-avr-lib, 01 2008.
16. A. K. Lenstra. Key lengths. Contribution to The Handbook of Information Security, 2004. http://cm.bell-labs.com/who/akl/key_lengths.pdf.
17. A. Liu and P. Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. Technical Report TR-2007-36, North Carolina State University, Department of Computer Science, November 2007.

18. M. Luk, A. Perrig, and B. Whillock. Seven cardinal properties of sensor network broadcast authentication. *Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, pages 147–156, 2006.
19. A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, 1996.
20. R. C. Merkle. A certified digital signature. In G. Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.
21. D. Naor, A. Shenhav, and A. Wool. One-time signatures revisited: Have they become practical. Cryptology ePrint Archive, Report 2005/442, 2005. `http://eprint.iacr.org/`.
22. J. P. Steinberger. The collision intractability of mdc-2 in the ideal-cipher model. In M. Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 34–51. Springer, 2007.
23. J. Viega. The AHASH Mode of Operation. *Manuscript available from http://www.cryptobarn.com/*, 2004.
24. S. Yu-long, M. Jian-feng, and P. Qing-qi. An Access Control Scheme in Wireless Sensor Networks. *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, pages 362–367, 2007.