A Performance Boost for Hash-based Signatures

Thomas Eisenbarth[†], Ingo von Maurich[‡], Christof Paar[‡], Xin Ye[†]

[†]Worcester Polytechnic Institute, Worcester, MA, USA
[‡]Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{teisenbarth,xye}@wpi.edu, {ingo.vonmaurich,christof.paar}@rub.de

Abstract. Digital signatures have become a key component of many embedded system solutions and are facing strong security and efficiency requirements. In this work, algorithmic improvements for the authentication path computation decrease the average signature computation time by close to 50 % when compared to state-of-the-art algorithms. The proposed scheme is implemented on an Intel Core i7 CPU and an AVR ATxmega microcontroller with optimized versions for the respective target platform. The theoretical algorithmic improvements are verified and cryptographic hardware accelerators are used to achieve competitive performance.

Keywords. hash-based cryptography, signatures, software, microcontroller, post-quantum cryptography, embedded security

1 Motivation

Today, digital signatures have become important for many embedded systems. Examples include firmware updates (a usage which is applicable to virtually every networked embedded system), identification (e.g., with smart cards), electronic payment (e.g., with smart phones via the NFC interface), or protection against product counterfeiting. Unfortunately, conventional digital signature schemes are very computational intensive and are a challenge on embedded microprocessors. Exploring signature schemes that are efficient on embedded platforms is thus of great interest and can offer advantage over the prevailing choices of (EC-)DSA and RSA.

Prior work by Rohde et al. [19] as well as by Hülsing et al. [11] suggest that the Merkle Signature Scheme (MSS) in combination with Winternitz One-Time Signatures (W-OTS) is a possible choice for a time-limited signature scheme and can be efficiently implemented in embedded systems. We analyze and extend the proposal by Rohde et al. and propose several modifications that lead to significant performance improvements.

Contribution We implement the proposed signature scheme on two wide-spread platforms (Intel Core i7 CPU and a low-cost AVR 8-bit microcontroller) and make use of available cryptographic hardware accelerators to gain maximum efficiency. Furthermore, we propose an improved algorithm for the authentication path computation of a Merkle tree. At the same time we decrease the average computation time by close to 50% compared to the most efficient authentication path computation algorithm at the price of a slightly increased memory consumption. Explicit formulas are developed to quantify the number of times each leaf of the Merkle tree is computed during the authentication path computation. The drawback of current authentication path computation algorithms is the unbalanced number of computations per leaf. Our improved algorithm mitigates this issue by reducing the number of computations for often used leaves and allows for more efficient computation of the authentication path.

2 Hash-Based Signatures

In the following we describe the foundations of the Merkle signature scheme. It was introduced in [18] and a detailed description of MSS can be found in [7]. Details about the implementation inspiring our work are given in [19]. We use Winternitz one-time signatures [8] for message signing. The one-time keys are generated using a PRNG to minimize storage requirements as proposed in [19].

The following components use an at least second preimage resistant, undetectable n-bit one-way function f and a cryptographic m-bit hash function g:

$$f : \{0,1\}^n \to \{0,1\}^n , \;\; g : \{0,1\}^* \to \{0,1\}^m$$

2.1 The Merkle Signature Scheme

Given a One-Time Signature Scheme (OTSS) a tree height H is chosen to allow for the creation of 2^H signatures that are verifiable with the same verification key. Let the nodes of the Merkle tree be denoted as $\nu_h[s]$ with $h \in \{0, \dots, H\}$ being the height of the node and $s \in \left\{0, \dots, 2^{H-h} - 1\right\}$ being the node index on height h.

Key Generation The 2^H leaves of the Merkle tree are defined to be digests $g(Y_i)$ of one-time verification keys Y_i. Starting from the leaves, the MSS verification key which is the root node of the Merkle tree $\nu_H[0]$ is generated following

$$\nu_{h+1}[i] = g\left(\nu_h[2i] \,||\, \nu_h[2i+1]\right), \;\; 0 \le h < H, \;\; 0 \le i < 2^{H-h-1},$$

meaning that a parent node is generated by hashing the concatenation of its two child nodes.

Signature Generation A Merkle signature $\sigma_s(d)$ of a digest $d = g(M)$ of a message M consists of a signature index s, a one-time signature σ_{OTS}, a one-time verification key Y_s, and an authentication path $(\textsc{Auth}_0, \dots, \textsc{Auth}_{H-1})$ that allows the verification of the one-time signature with respect to the public MSS verification key, hence

$$\sigma_s(d) = \left(s, \sigma_{\text{OTS}}, Y_s, (\textsc{Auth}_0, \dots, \textsc{Auth}_{H-1})\right).$$

The signature index $s \in \left\{0, \ldots, 2^H - 1\right\}$ is incremented with every issued signature. The OTSS is applied using signature key X_s to generate the signature $\sigma_{\mathrm{OTS}} = \mathsf{Sign}_{\mathrm{OTS}}(d, X_s)$ of the message digest d. The authentication path for the sth leaf are all sibling nodes $\textsc{Auth}_h$, $h \in \{0, \ldots, H-1\}$ on the path from leaf $\nu_0\,[s]$ to the root node $\nu_H\,[0]$. It enables the verifier to recompute the root node of the Merkle tree and authenticates the current one-time signature.

We would like to stress that the signature generation reflects the structure of an online/offline signature scheme. The authentication path only depends on the OTSS verification key Y_s which is known prior to the message and hence can be precomputed.

Signature Verification. Given a message digest $d = g\,(M)$ and a signature $\sigma_s\,(d)$ the verifier checks the one-time signature σ_{OTS} with the underlying one-time signature verification algorithm $\mathsf{Verify}_{\mathrm{OTS}}\,(d, \sigma_s(d))$. In addition, the root node is reconstructed using the provided authentication path

$$\phi_{h+1} = \begin{cases} g\,(\phi_h \,||\, \textsc{Auth}_h)\,, \text{ if } \lfloor s/2^h \rfloor \equiv 0 \bmod 2 \\ g\,(\textsc{Auth}_h \,||\, \phi_h)\,, \text{ if } \lfloor s/2^h \rfloor \equiv 1 \bmod 2 \end{cases}, \quad \phi_0 = \nu_0\,[s]\,, \quad h = 0, \ldots, H-1.$$

If the one-time signature σ_{OTS} is successfully verified and ϕ_H is equal to $\nu_H\,[0]$ the MSS signature is accepted.

2.2 Winternitz One-Time Signatures

Winternitz OTS [8] are a convenient choice for the one-time signature scheme, as they reduce the overall signature length. The Winternitz parameter $w \geq 2$ determines how many bits are signed simultaneously and t determines of how many random n-bit strings x_i the Winternitz signature keys consist.

$$t = t_1 + t_2, \;\; t_1 = \left\lceil \frac{n}{w} \right\rceil, \;\; t_2 = \left\lceil \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rceil$$

Key Generation A W-OTS signature key $X = (x_0, \ldots, x_{t-1})$ is generated by selecting t random bit strings $x_i \in \{0,1\}^n$, $0 \leq i < t$. The W-OTS verification key $Y = g\,(y_0 \,||\, \ldots \,||\, y_{t-1})$ is computed from the signature key by applying f $2^w - 1$ times to each x_i giving $y_i = f^{2^w - 1}\,(x_i)\,,\; 0 \leq i < t$ and computing the hash of the concatenated y_i's. Note, the superscript denotes multiple executions of f, e.g., $f^2\,(x_i) = f\,(f\,(x_i))$ and $f^0\,(x_i) = x_i$.

Signature Generation A signature for a message M is created by signing its digest $d = g\,(M)$ under key X. Digest d is divided into t_1 blocks $b_0, \ldots, b_{t_1 - 1}$ of length w and a checksum $c = \sum_{i=0}^{t_1 - 1} (2^w - b_i)$ is computed. Checksum c is divided into t_2 blocks $b_{t_1}, \ldots, b_{t-1}$ of length w (zero-padding to the left is applied if c or d are no multiples of w). The W-OTS signature $\sigma_{\text{W-OTS}} = (\sigma_0, \ldots, \sigma_{t-1})$ is computed with $\sigma_i = f^{b_i}\,(x_i)\,,\; 0 \leq i < t$.

Signature Verification Given a message digest $d = g(M)$, a signature $\sigma_{\text{W-OTS}}$ and a verification key Y_s the verifier generates blocks $b_0, \ldots, b_{t-1}$ from d as in signature generation and reconstructs

$$Y'_s = g\left(f^{2^w-1-b_0}(\sigma_0) \,||\, \ldots \,||\, f^{2^w-1-b_{t-1}}(\sigma_{t-1})\right).$$

If Y'_s equals Y_s the signature is valid, otherwise it has to be rejected. When using W-OTS signatures in MSS, transmitting Y_s and comparing Y_s to Y'_s can be omitted. Y'_s can simply be used together with the nodes of the authentication path to recompute the root of the Merkle tree. If the recomputed root equals the MSS public key, then Y'_s is a valid OTS verification key.

2.3 Private Key Generation

Storing 2^H one-time signature or verification keys can be an infeasible task, especially on constrained implementation platforms. Generating keys on-the-fly by using a PRNG significantly reduces the required storage space (cf. [19]).

Each W-OTS signature key $X_i = (x_0, \ldots, x_{t-1})$, $0 \le i < 2^H$ is generated by the PRNG from a seed $\textsc{Seed}_{\text{W-OTS}_i}$. These seeds in turn are also generated by the PRNG from a initial randomly selected seed $\textsc{Seed}_0 \in_R \{0,1\}^n$ which serves as the MSS signature key. On input of k_{i} the PRNG outputs a random string r_{i+1} and an updated seed k_{i+1}.

$$\textsc{Prng} : \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n, k_i \to (k_{i+1}, r_{i+1}) \qquad (1)$$

Starting from the initial $\textsc{Seed}_0$ the seeds for the signature keys $\textsc{Seed}_{\text{W-OTS}_i}$ are created by

$$(\textsc{Seed}_{i+1}, \textsc{Seed}_{\text{W-OTS}_i}) \leftarrow \textsc{Prng}(\textsc{Seed}_i), \; 0 \le i < 2^H.$$

The t n-bit strings of the i-th W-OTS signature key $X_i = (x_0, \ldots, x_{t-1})$, $0 \le i < 2^H$ are then generated by

$$(\textsc{Seed}_{\text{W-OTS}_i}, x_j) \leftarrow \textsc{Prng}(\textsc{Seed}_{\text{W-OTS}_i}), \; 0 \le j < t.$$

2.4 Authentication Path Computation

Creating an authentication path for a specific leaf s can be accomplished by storing all tree nodes in memory and looking up the required nodes when needed. However, because of the exponential growth of nodes in tree height H this approach becomes infeasible for reasonable practical applications. Hence, algorithms for efficient on-the-fly authentication path computation during signature generation are required.

The currently best known algorithm for on-the-fly computation of authentication nodes is the BDS algorithm [7] (Algo. 3, cf. Appendix). It makes use of several treehash algorithm instances $\textsc{Treehash}_h$ for heights $0 \le h \le H-K-1$. The treehash algorithm was introduced in [18] and modified in [21]. It allows to efficiently create (parts of) Merkle trees. In the BDS algorithm each instance is initialized with a leaf index s to which it computes the corresponding node value. Each instance is updated until the required authentication node is computed. During a treehash update the next leaf is created and parent nodes are computed if possible.

The generation of the authentication path is split up into two parts that go alongside with the key and signature generation of MSS. During key generation all treehash instances $\textsc{Treehash}_h$ are initialized with $\nu_h[3]$ and the first authentication path stored is $\textsc{Auth}_h = \nu_h[1]$, $0 \leq h \leq H-1$.

The BDS algorithm generates left authentication nodes either by computing the leaf value or by one hash-function evaluation of the concatenation of two previously computed nodes that are held in memory. Right authentication nodes in contrast are computed from the leaf up, which is computationally more expensive. Since right nodes close to the top are expensive to compute a positive integer $K \geq 2, (H-K$ even) decides how many of these nodes are stored in $\textsc{Retain}_h, H-K \leq h \leq H-2$ during key generation.

Authentication nodes change every 2^h steps for height h. During signature generation the treehash instances are updated and if a authentication node from a treehash instance is used, the instance is re-initialized to compute the next authentication node for that height.

2.5 Security of MSS

The security properties of the signature scheme described above is discussed in [7]. Specifically, the work shows that the Lamport-Diffie one-time signatures [14] are existentially unforgeable under an adaptive chosen message attack (i.e., CMA-secure), if the chosen one-way function is preimage resistant. The employed Merkle signature scheme is also CMA-secure if the underlying OTS is CMA-secure and if the underlying hash function is collision resistant. For increased efficiency (and shorter signatures) we chose Winternitz OTS rather than the classic Lamport-Diffie OTS. The security of the Winternitz one-time signatures is discussed in [5, 8, 10]. The findings in [5] and [10] show that Winternitz OTS are CMA-secure if used with pseudo-random functions or collision-resistant, undetectable one-way functions, respectively. The level of bit security lost by using a small Winternitz-parameter is in both cases rather small. In our case, the biggest Winternitz parameter is $w = 4$, hence we still provide a security level of approx. 95 bits for a 128-bit PRF or 116 bits for W-OTS+ [10]). Related discussions for a similar MSS scheme can also be found in [6].

3 Optimized Authentication Path Computation

Since the Merkle-tree is not stored, the parts of the Merkle tree needed for the authentication path must be generated. One optimized algorithm for this purpose is the BDS algorithm [7]. Its design goal was to minimize costly leaf computations. In the following we describe further optimizations that, at the expense of a slightly higher memory consumption, reduce the number of computations for the leaves that are computed most often in the BDS algorithm. We thereby reduce the overall number of leaf calculations and hence the overall computation time by close to 50%.

3.1 Authentication Path Computation

The authentication path consists of nodes of the Merkle tree. For the computation of upcoming authentication nodes we use several stacks of nodes for different heights of the tree. Treehash instances $\textsc{Treehash}_h$ are used for heights $0 \le h \le H-K-1$. Each instance is initialized with a leaf index s and is updated in Algo. 3 until the required authentication node is computed. During a treehash update the next leaf is created and parent nodes are computed by hashing previously created nodes if possible. Authentication nodes change every 2^h steps for height h and if an authentication node is used from a treehash instance, this instance is re-initialized to compute the following authentication node for that height.

Preliminaries The total number of leaf computations that occur during execution of Algo. 3 can be calculated by counting all invocations of $\textsc{Leafcalc}$, a function that on input s outputs leaf $\nu_0[s]$. As mentioned in [7] it is possible to omit $\textsc{Leafcalc}$ in Step 3 of Algo. 3 since the sth W-OTS key pair is used to sign the current message, hence the verification key can be computed from the signature and one additional hash computation yields leaf $\nu_0[s]$. If a different OTSS is used the verification key is part of the OTS and can be hashed to create $\nu_0[s]$. This saves 2^{H-1} $\textsc{Leafcalc}$ invocations. Careful analysis of Algo. 3 leads to the total number of leaf computations in the BDS algorithm

$$N_{H,K_{\text{total}}} = \sum_{h=0}^{H-K-1} \left(2^{H-1} - 2^{h+1}\right) = (H-K)\,2^{H-1} - 2^{H-K+1} + 2.$$

In order to count the necessary computations for a specific leaf s during execution of Algo. 3 we have to consider all occurrences of s as parameter of $\textsc{Leafcalc}$, except for when s is a left leaf (Step 3 of Algo. 3), as explained above. To determine if leaf s is computed in treehash instance $\textsc{Treehash}_h$ we make the following observation: $\textsc{Treehash}_0$ computes leaves $(5), (7), (9), \ldots$, $\textsc{Treehash}_1$ computes leaves $(10, 11), (14, 15), \ldots$, $\textsc{Treehash}_2$ computes leaves $(20, 21, 22, 23), (28, 29, 30, 31), \ldots$ and so forth. Hence, the total number of computations for leaf s is given by

$$N_{H,K}(s) = \sum_{h=0}^{H-K-1} \left\lfloor \frac{s \bmod 2^{h+1}}{2^h} \right\rfloor \cdot \left\lceil \frac{\left\lfloor \frac{s}{5\cdot 2^h} \right\rfloor}{2^H} \right\rceil$$

Drawbacks A drawback of the BDS algorithm (Algo. 3) is that it does not balance the computation of leaf nodes. There are leaves that are calculated various times, while others are barely touched. On average each leaf of the Merkle tree is computed $\overline{N_{H,K}} = N_{H,K_{\text{total}}}/2^H \approx \frac{1}{2}(H-K)$ times. However, the computations per leaf deviate from the average as shown in Fig. 1 for a Merkle tree $(H=10, K=2)$ with 1024 leaves.

3.2 Balanced Authentication Path Computation

Since the rightmost nodes of each treehash instance are calculated most frequently, we propose to cache and reuse them for balancing the leaf computations. We use an array RIGHTNODES to store those nodes. Note, the root of each treehash instance and the complete treehash instance TREEHASH_0 are not stored since lower treehash instances do not require those nodes. This leads to a significantly reduced computation time, at the cost of an increased memory consumption.

From TREEHASH_1 we store node $\nu_0\,[7]$, from TREEHASH_2 we store nodes $\nu_1\,[7]$ and $\nu_0\,[15]$ and so on. More generally, we store h nodes $\nu_j\left[2^{2+h-j}-1\right], j = 0,\dots,h-1$ for each instance TREEHASH_h, $1 \le h \le H-K-1$. The required storage space is

$$S_{\text{RightNodes}}(H,K) = \sum_{h=1}^{H-K-1} h = \binom{H-K}{2} = \triangle_{H-K-1}.$$

Table 1 lists the storage requirements for common $H-K$ values. The initialization of the RIGHTNODES array is done during the computation of the public key of the Merkle tree. The updated initial setup is formalized in Algo. 2.

Table 1. Storage space required by the RIGHTNODES array where the rightmost nodes of each treehash instance TREEHASH_h, $h = 1,\dots,H-K-1$ are stored for reusage by lower treehash instances.

$H-K$	$\triangle_{H-K-1}$	128-bit digest	160-bit digest	256-bit digest
6	15	240 byte	300 byte	480 byte
8	28	448 byte	560 byte	896 byte
10	45	720 byte	900 byte	1440 byte
12	66	1056 byte	1320 byte	2112 byte
14	91	1456 byte	1820 byte	2912 byte
16	120	1920 byte	2400 byte	3840 byte
18	153	2448 byte	3060 byte	4896 byte

In Step 5 of Algo. 3 the treehash instances receive updates if they are initialized and not finished. In every update one leaf is computed and higher nodes are generated if possible by hashing concatenated nodes from the stack. During the last update before the treehash instance is finished, the rightmost leaf of this treehash instance is computed and all other rightmost nodes of this treehash instance are consecutively generated. If the leaf index $s \equiv 2^h - 1 \mod 2^h$ in instance $\textsc{Treehash}_h$, we store the following nodes in the $\textsc{Rightnodes}$ array starting from offset $h(h-1)/2$. An adapted version of the treehash update algorithm is given in Algo. 1.

In every second re-initialization of treehash instances $\textsc{Treehash}_h$, $h = 0, \ldots, H-K-2$ the authentication node can be copied from the $\textsc{Rightnodes}$ array because it has been computed before by treehash instance $\textsc{Treehash}_{h+1}$. If $s+1 \equiv 0 \mod 2^{h+2}$ the authentication node can be copied from the $\textsc{Rightnodes}$ array and if $s+1 \equiv 2^{h+1} \mod 2^{h+2}$ the authentication node has to be computed. If we can reuse nodes, we not only copy the authentication node (root of $\textsc{Treehash}_h$) but also its rightmost child nodes from $\textsc{Rightnodes}$, so they can be reused for instances $\textsc{Treehash}_j$, $j < h$. This improvement can be easily integrated into the BDS algorithm by modifying Step 4c) accordingly.

Comparison In order to quantify our improvements, we give the total amount of leaf computations and show how to determine the leaf computations for a specific leaf s. As before, each instance $\textsc{Treehash}_h$ computes 2^h leaves until they are finished. The re-initializations however are halved for treehash instances $\textsc{Treehash}_h$, $h = 0, \ldots, H-K-2$, to $2^{H-h-2} - 1$ re-initializations because in half of all cases previously computed nodes can be copied from the $\textsc{Rightnodes}$ array and the $\textsc{Leafcalc}$ computations are skipped. Hence, the number of calls to $\textsc{Leafcalc}$ from each $\textsc{Treehash}_h$ instance is $2^{H-2} - 2^h$. The treehash instance $\textsc{Treehash}_{H-K-1}$ cannot copy nodes from higher instances since it is the topmost treehash instance. It calls $\textsc{Leafcalc}$ as before, resulting in $2^{H-1} - 2^{H-K}$ computations. The total number of leaf computations is

$$\begin{aligned} N'_{H,K_{\text{total}}} &= \sum_{h=0}^{H-K-2} \left(2^{H-2} - 2^h\right) + 2^{H-1} - 2^{H-K} \\ &= (H-K+1)\, 2^{H-2} - 3 \cdot 2^{H-K-1} + 1. \end{aligned}$$

When compared to $N_{H,K_{\text{total}}}$ of the BDS algorithm this is nearly a 50% reduction.

To retrieve the number of leaf computations in the improved version for a specific leaf s we have to check whether s is a left or a right leaf. If s is even, it is a left leaf and can be computed from the current one-time signature or verification key as mentioned in Section 3.1 for Step 3 of Algo. 3. If s is odd, it is a right leaf thus $\textsc{Leafcalc}$ is not executed directly. To determine if s is computed in treehash instance $\textsc{Treehash}_h$, $h = 0, \ldots, H-K-2$, we have to consider that in half of all cases it is copied and not computed. For this purpose we construct function $\delta'_{H,K}(s)$ that returns the number of times leaf s is computed in treehash instances $\textsc{Treehash}_h, h = 0, \ldots, H-K-2$.

$$\delta'_{H,K}(s) = \sum_{h=0}^{H-K-2} \left\lfloor \frac{s \bmod 2^{h+1}}{2^h} \right\rfloor \cdot \left\lceil \frac{\left\lfloor \frac{s}{5 \cdot 2^h} \right\rfloor}{2^H} \right\rceil \cdot \left(1 - \left\lfloor \frac{s \bmod 2^{h+2}}{2^{h+1}} \right\rfloor\right)$$

The topmost treehash instance $\textsc{Treehash}_{H-K-1}$ cannot copy nodes from the $\textsc{Rightnodes}$ array because the required nodes have not been computed so far. Thus, we have to count the number of computations for this instance as in the unoptimized version. The total number of times leaf s is generated during the computation of all authentication nodes can now be summed up to

$$N'_{H,K}(s) = \left\lfloor \frac{s \bmod 2^{H-K}}{2^{H-K-1}} \right\rfloor \cdot \left\lceil \frac{\left\lfloor \frac{s}{5 \cdot 2^{H-K-1}} \right\rfloor}{2^H} \right\rceil + \delta'_{H,K}(s) \, .$$

On average each leaf is now computed $\overline{N'_{H,K}} = N'_{H,K_{\text{total}}}/2^H \approx \frac{1}{4}(H-K+1)$ times. The reduced number of computations for each leaf is shown in Fig. 2. Visual comparison between Fig. 1 and Fig. 2 already gives an intuition of the reduction and balancing of leaf computations. For further comparisons see Fig. 3 in the appendix. Table 2 compares the total number of leaf computations, how

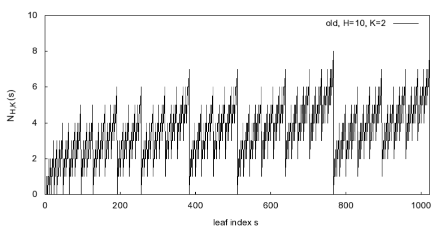

Fig. 1. Number of times each leaf is computed by the original BDS algorithm for a Merkle tree of height $H = 10$ and $K = 2$.

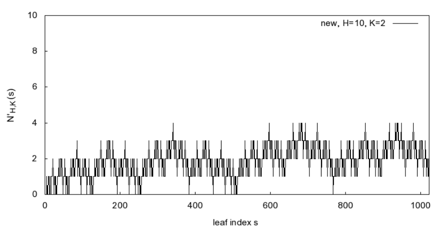

Fig. 2. Number of times each leaf is computed by our variation for a Merkle tree of height $H = 10$ and $K = 2$.

often a leaf has to be computed in the worst-case, and the average number of leaf computations for common heights $H = \{10, 16, 20\}$ and $K = \{2, 4\}$. The total number of leaf computations as well as the average computations per leaf are decreased by about $38 - 48\%$ for the chosen parameters of H and K. Both the worst-case computation time as well as the average signature computation time are decreased. This means that, for example, battery-powered devices greatly profit from the reduced overall computation time which directly relates to the overall power consumption.

Since all but the topmost treehash instance only need to be computed every second time, the number of updates per signature (Algo.3, Step 5) can be reduced from $\lceil (H-K)/2 \rceil$ to $\lceil (H-K+1)/4 \rceil$. As a result, the average update time is much better balanced than in Algo. 3 and the worst case computation time is also improved. The BDS algorithm needs to store $3H + \lfloor H/2 \rfloor - 3K + 2^K - 2$ tree nodes and $2(H-K)+1$ PRNG seeds as signature key. Due to storing the rightmost nodes our improved algorithm increases the number of tree nodes that have to be stored by $\binom{H-K}{2}$. Even if the additional memory is used to increase K for the original BDS algorithm, the speedup is still significant. E.g., comparing

Table 2. Overview of the necessary computations for a Merkle tree with parameters H and K when executing Algo. 3. Furthermore, the worst-case computations for a leaf is listed together with the average computations $\overline{N_{H,K}}$ and $\overline{N'_{H,K}}$. The variance of $N_{H,K}(s)$ and $N'_{H,K}(s)$ is denoted by $\sigma^2_{H,K}$ and $\sigma'^2_{H,K}$.

H	K	$N_{H,K_{\text{tot}}}$	$N'_{H,K_{\text{tot}}}$	$\overline{N_{H,K}}$	$\overline{N'_{H,K}}$	%	$\sigma^2_{H,K}$	$\sigma'^2_{H,K}$	%	max. $N_{H,K}(s)$	max. $N'_{H,K}(s)$	%
10	2	3586	1921	3.50	1.88	46.4	2.24	0.73	67.3	8	4	50.0
10	4	2946	1697	2.88	1.66	42.4	1.60	0.50	68.5	6	3	50.0
10	6	2018	1257	1.97	1.23	37.7	1.02	0.33	67.9	4	2	50.0
16	2	425986	221185	6.50	3.38	48.1	3.75	1.11	70.4	14	7	50.0
16	4	385026	206849	5.88	3.16	46.3	3.11	0.88	71.6	12	6	50.0
16	6	325634	178689	4.97	2.73	45.1	2.53	0.71	72.1	10	5	50.0
20	2	8912898	4587521	8.50	4.38	48.5	4.75	1.36	71.4	18	9	50.0
20	4	8257538	4358145	7.88	4.16	47.2	4.11	1.13	72.5	16	8	50.0
20	6	7307266	3907585	6.97	3.73	46.5	3.53	0.96	72.9	14	7	50.0

our $(H, K) = (16, 4)$ to BDS(16, 6) gives comparable storage requirements, but still a speedup of 36%. The verification key and signature sizes remain unaffected: the verification key size is m and the signature size remains at $t \cdot n + H \cdot m$.

4 Implementation and Results

In the following we describe our choices for the cryptographic primitives which we use to implement the proposed signature scheme described in Sections 2 and 3. We then detail on the target platforms and give performance figures for key and signature generation as well as signature verification.

4.1 A High Performance Merkle Signature Engine

We implemented two versions with different *hash functions g* for the Merkle tree. Both versions use AES-128 in an MJH construction [15]. Using AES-128 as block cipher is favorable from a performance perspective as existing AES co-processors can be used. MJH is collision resistant for up to $\mathcal{O}\left(2^{\frac{2n}{3}-\log n}\right)$ queries when instantiated with a n-bit ideal block cipher. With AES-128 as an ideal cipher[1], this results in 80 bits collision resistance [15]. On the downside, MJH produces 256-bit hash outputs which in the MSS setting leads to an increased key and signature size. Hence, we also implement a version that shortens the 256-bit output of MJH to 160-bit, resulting in smaller key and signature sizes. This also reduces the number of times the AES-engine needs to be called when

[1] Please note that with recent cryptanalytic breakthroughs against the AES family of ciphers [4], it is doubtful that AES-128 is indistinguishable from an ideal block cipher

creating nodes in the Merkle tree. *One-way function* f is implemented based on AES-128 in an MMO [16, 17] construction: $f(x_i) := \mathrm{AES}_{\mathrm{IV}}(x_i) \oplus x_i$. The *PRNG* defined in (1) is implemented based on the leakage-2-limiting PRNG proposed in [20]. In particular, $\mathrm{PRNG}(k_i) := (\mathrm{AES}_{k_i}(0^{128}), \mathrm{AES}_{k_i}(0^{127}||1))$, where AES_{k_i} denotes the AES-128 with a 128-bit key k_i. A discussion of the leakage properties of this implementation can be found in [9].

4.2 Implementation Platforms

We implement the signature scheme on two different platforms. On the one side we choose a lightweight and low-cost 8-bit Atmel ATxmega microcontroller and on the other side a powerful Intel Core i7 notebook CPU.

Intel Core i7-2620M 64-bit CPU Intel's off-the-shelf Core i7-2620M 64-bit Sandy Bridge notebook CPU [12] features two cores running at 2.70 GHz (with Turbo Boost technology up to 3.40 GHz). For accurate measurement, we disabled Turbo Boost and hyper-threading during our benchmarks. The CPU incorporates the recent extensions to the x86 instruction set that improve the performance when en-/decrypting data using AES. The extension is called AES-NI and consists of six additional instructions [13]. All standardized key lengths (128 bit, 192 bit, 256 bit) are supported for a block size of 128 bit.

Atmel AVR ATxmega128A1 8-bit Microcontroller We are using the Atmel evaluation board AVR XPLAIN [3] that features an ATxmega128A1 microcontroller [1, 2], as it is a typical example for a low-power embedded platform.. The ATxmega offers hardware accelerators for DES and AES and is clocked at 32 MHz. The hardware acceleration is limited to AES with 128-bit key and block size.

4.3 Performance Results

In the following we give performance figures of the signature scheme for selected Merkle tree heights H and parameters K and w on both platforms.

CPU Performance On the Intel CPU we measure the time it takes to create the root node of the Merkle tree, i.e., the verification key generation. We iterate over all leaves and sign random messages to measure the average computation time that is needed to create a valid MSS signature. Additionally, we measure the time it takes to verify an MSS signature. Signature computation includes creating the signing key, performing a one-time signature with the created signing key, and generating the next authentication path (the last step can be removed, as it can be precomputed at any time between two signing operations). The measurement is done for tree height $H = 16$ with $K = 2$ and $w = 2$. Note, due to the binary tree structure the root node computation can be parallelized if more than one CPU core is available, which would bring down the required computation time by roughly the factor of cores used. We compare our results against the originally proposed signature scheme [19] in Table 3.

Table 3. Performance figures of a Merkle tree with parameters $H = 16, K = 2, w = 2$ on an Intel i7 CPU and $H = 10, K = 2, w = 2$ on an ATxmega microcontroller. f is implemented using a hardware-accelerated AES-128 (AES-NI instructions, ATxmega crypto accelerator) in MMO construction. g is implemented using AES-128 in an MJH-256 construction and with the output truncated to 160 bit. The Intel CPU was clocked at 2.7 GHz and the ATxmega at 32 MHz.

Hash g		MJH-256 w/ AES-128			MJH-160 w/ AES-128		
Target		[19]	our	impr.	[19]	our	impr.
Core i7	KeyGen	6546.9 ms	6037.5 ms	8%	4218.7 ms	3886.3 ms	8%
Core i7	Sign	743.9 us	401.3 us	46%	487.1 us	256.2 us	47%
Core i7	Verify	76.1 us	78.1 us	-3%	50.8 us	49.3 us	3%
AVR	Sign	110.0 ms	64.9 ms	41%	70.7 ms	41.7 ms	41%
AVR	Verify	18.4 ms	18.4 ms	0%	11.0 ms	11.0 ms	0%

Compared to the previous results of [19] our improved algorithm in combination with the exchanged PRNG yields on average a performance gain of 46-47 % for signature generation. The new PRNG improves the computation time on average by 8%, the algorithmic changes to the authentication path computation algorithm yield 38-39% points.

When generating verification keys an 8% improvement can be observed. This is due to the exchanged PRNG which uses a hardware-accelerated AES-engine since our algorithmic improvements do not affect key generation. Signature verification is more or less stable, regardless of cipher/algorithm combinations and is about a factor of 5 faster than signature generation.

Microcontroller Performance On the microcontroller we measure the average computation time that is needed to create a valid MSS signature (including next authentication path computation) and the time it takes to verify an MSS signature. We omit the verification key generation since for reasonable tree heights it is an infeasible task for the microcontroller. Verification keys have to be computed once on a computer platform when initializing the microcontroller. The code was compiled using `avr-gcc` version 3.3.0. We found optimization stage `-O2` to provide the best tradeoff between runtime and code size.

The results on the microcontroller are in accordance with the results observed on the Intel CPU. The average signature generation time improves by 41 % when using our proposed changes. Signature verification remains stable and is four times faster than signature generation. The memory consumption is listed in

Table 4. Compared to the setting of [19] we need more flash and SRAM memory due to the additional storage for the Rightnodes array.

Table 4. Required memory on the ATxmega128A1 microcontroller. In total 128 kByte flash memory and 8 kByte SRAM are available on this device. Memory consumption is reported in bytes and includes the verification and signature keys.

		MJH-256 w/ AES-128				MJH-160 w/ AES-128			
		[19]		our		[19]		our	
H	K	Flash	SRAM	Flash	SRAM	Flash	SRAM	Flash	SRAM
10	2	10,608	1,486	12,070	2,382	10,204	1,066	11,352	1,626
10	4	10,726	1,604	11,768	2,084	10,250	1,112	11,138	1,412
10	6	11,994	2,874	12,752	3,066	11,018	1,878	11,726	1,998

Table 5 compares key and signature sizes for different MSS implementations. Note that the increased signature sizes for [11] enable on-card key generation.

Table 5. Comparison of signing key (sk), verification key (vk), and signature size (sig) between [19], our improvement, and XMSS$^+$ [11] for common (H, K, w) parameter sets. All sizes are reported in bytes.

			MJH-256			MJH-160			[19] (MJH-256)			[19] (MJH-160)			XMSS$^+$ [11]		
H	K	w	sk	vk	sig	sk	vk	sig	sk	vk	sig	sk	vk	sig	sk	vk	sig
16	2	2	5,335	32	2,640	3,547	20	1,680	2,423	32	2,640	1,727	20	1,680	3,760	544	3,476
16	2	4	5,335	32	1,584	3,547	20	1,008	2,423	32	1,584	1,727	20	1,008	3,200	512	1,892
20	4	2	7,049	32	2,768	4,649	20	1,760	3,209	32	2,768	2,249	20	1,760	4,303	608	3,540
20	4	4	7,049	32	1,712	4,649	20	1,088	3,209	32	1,712	2,249	20	1,088	3,744	576	1,956

5 Conclusion

We presented a novel algorithmic improvement for authentication path computation in MSS that balances leaf computations. The proposed improvements have been implemented on two platforms and were compared to previous proposed algorithms showing significant improvements. Furthermore, we gave explicit formulas to quantify the number of leaf computations when using MSS.

We stated theoretically achievable performance gains and verified them practically. The algorithmic improvement decreases the required computation time for signature creation in theory as well as in practice. The performance figures show that Merkle signatures are not only practical, but also resource-friendly and fast. As such they are an advantageous choice for, e.g., digital signature smart cards.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant No. 1261399 and by grant 01ME12025 SecMobil of the German Federal Ministry of Economics and Technology.

References

1. Atmel. ATxmega128A1 Data Sheet. `http://www.atmel.com/dyn/resources/prod_documents/doc8067.pdf`.
2. Atmel. AVR XMEGA A Manual. `http://www.atmel.com/dyn/resources/prod_documents/doc8077.pdf`.
3. Atmel. AVR XPLAIN board. `http://www.atmel.com/dyn/resources/prod_documents/doc8203.pdf`.
4. A. Biryukov, D. Khovratovich, and I. Nikoli. Distinguisher and related-key attack on the full aes-256. In S. Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 231–249. Springer Berlin Heidelberg, 2009.
5. J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert. On the Security of the Winternitz One-Time Signature Scheme. In A. Nitaj and D. Pointcheval, editors, *Progress in Cryptology AFRICACRYPT 2011*, volume 6737 of *Lecture Notes in Computer Science*, pages 363–378. Springer Berlin / Heidelberg, 2011.
6. J. Buchmann, E. Dahmen, and A. Hülsing. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In B.-Y. Yang, editor, *PQCrypto*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011.
7. J. Buchmann, E. Dahmen, and M. Szydlo. Hash-based Digital Signature Schemes. In D. J. Bernstein, J. Buchmann, and E. Dahmen, editors, *Post-Quantum Cryptography*, pages 35–93. Springer Berlin Heidelberg, 2009.
8. C. Dods, N. P. Smart, and M. Stam. Hash Based Digital Signature Schemes. In *Cryptography and Coding*, pages 96–115. Springer, 2005.
9. T. Eisenbarth, I. von Maurich, and X. Ye. Faster Hash-based Signatures with Bounded Leakage. In *Selected Areas in Cryptography — SAC 2013*, August 2013.
10. A. Hülsing. W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *AFRICACRYPT*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013.
11. A. Hülsing, C. Busold, and J. Buchmann. Forward Secure Signatures on Smart Cards. In L. R. Knudsen and H. Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2012.
12. Intel. Intel Core i7 2620M Specifications. `http://ark.intel.com/products/52231/Intel-Core-i7-2620M-Processor-(4M-Cache-2_70-GHz)`.
13. Intel. Whitepaper on the Intel AES Instructions Set. `http://software.intel.com/file/24917`.
14. L. Lamport. Constructing Digital Signatures from a One-Way Function. Technical report, CSL-98, SRI International, 1979.
15. J. Lee and M. Stam. MJH: A Faster Alternative to MDC-2. In A. Kiayias, editor, *Topics in Cryptology CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 213–236. Springer Berlin / Heidelberg, 2011.

16. S. M. Matyas, C. H. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.
17. A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC, 1997. Algorithm 9.41.
18. 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.
19. S. Rohde, T. Eisenbarth, E. Dahmen, J. Buchmann, and C. Paar. Fast Hash-Based Signatures on Constrained Devices. In *Smart Card Research and Advanced Applications — CARDIS 2008*, pages 104–117. Springer, 2008.
20. F.-X. Standaert, O. Pereira, Y. Yu, J.-J. Quisquater, M. Yung, and E. Oswald. Leakage Resilient Cryptography in Practice. In A.-R. Sadeghi, D. Naccache, D. Basin, and U. Maurer, editors, *Towards Hardware-Intrinsic Security*, Information Security and Cryptography, pages 99–134. Springer Berlin Heidelberg, 2010.
21. M. Szydlo. Merkle Tree Traversal in Log Space and Time. In C. Cachin and J. Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554. Springer, 2004.

A Appendix

Algorithm 1 Improved treehash update

```
Input: Height $h$, current index s, RIGHTNODES array
Output: updated RIGHTNODES array, updated Treehash instance $\text{TREEHASH}_h$
  Compute the $s$th leaf: $\text{NODE}_1 \leftarrow \text{LEAFCALC}(s)$
  if $s \equiv 2^h - 1 \left(\bmod 2^h\right)$ and $\text{NODE}_1$.height() $< h$ then
    offset $= h\,(h-1)\,/2$
    RIGHTNODES[offset] $\leftarrow \text{NODE}_1$
  end if
  while $\text{NODE}_1$ has the same height as the top node on $\text{TREEHASH}_h$ do
    Pop the top node from the stack: $\text{NODE}_2 \leftarrow \text{TREEHASH}_h.pop()$
    Computer their parent node: $\text{NODE}_1 \leftarrow g(\text{NODE}_2 \| \text{NODE}_1)$
    if $s \equiv 2^h - 1 \left(\bmod 2^h\right)$ then
      offset = offset + 1
      RIGHTNODES[offset] $\leftarrow \text{NODE}_1$
    end if
  end while
  Push the parent node on the stack: $\text{TREEHASH}_h.push(\text{NODE}_1)$
```

Algorithm 2 Key generation and initial setup for the improved traversal algorithm.

```
Input: $H, K$
Output: Public key $\nu_H[0]$, Authentication path, RIGHTNODES array, TREEHASH stacks, RETAIN stacks
1: Public Key
   Calculate and publish tree root, $\nu_H[0]$.
2: Initial Right Nodes
   $i = 0$
   for $h = 1$ to $H - K - 1$ do
     for $j = 0$ to $h - 1$ do
       Set RIGHTNODES$[i] = \nu_j\left[2^{2+h-j} - 1\right]$.
       $i = i + 1$
3: Initial Authentication Nodes
   for each $h \in \{0, 1, \ldots, H - 1\}$ do
     Set AUTH$_h = \nu_h[1]$.
4: Initial Treehash Stacks
   for each $h \in \{0, 1, \ldots, H - K - 1\}$ do
     Setup TREEHASH$_h$ stack with $\nu_h[3]$.
5: Initial Retain Stacks
   for each $h \in \{H - K, \ldots, H - 2\}$ do
     for each $j \in \left\{2^{H-h-1}, \ldots, 0\right\}$ do
       RETAIN$_h$.push($\nu_h[2j + 3]$).
```

Algorithm 3 Algorithm for authentication path computation as presented in [7]

```
Input: $s \in \left\{0, \ldots, 2^H - 2\right\}$, $H, K$, and the algorithm state.
Output: Authentication path $A_{s+1}$ for leaf $s + 1$.
1: Let $\tau = 0$ if leaf $s$ is a left node or let $\tau$ be the height of the first parent of leaf $s$ which is a left
   node: $\tau \leftarrow \max\{h : 2^h | (s + 1)\}$
2: If the parent of leaf $s$ on height $\tau + 1$ is a left node, store the current authentication node on
   height $\tau$ in KEEP$_\tau$:
   if $\lfloor s/2^{\tau+1} \rfloor$ is even and $\tau < H - 1$ then KEEP$_\tau \leftarrow$ AUTH$_\tau$
3: If leaf $s$ is a left node, it is required for the authentication path of leaf $s + 1$:
   if $\tau = 0$ then AUTH$_0 \leftarrow$ LEAFCALC($s$)
4: Otherwise, if leaf $s$ is a right node, the auth. path for leaf $s + 1$ changes on heights $0, \ldots, \tau$:
   if $\tau > 0$ then
   a) The authentication path for leaf $s + 1$ requires a new left node on height
      $\tau$. It is computed using the current authentication node on height $\tau - 1$
      and the node on height $\tau - 1$ previously stored in KEEP$_{\tau-1}$. The node
      stored in KEEP$_{\tau-1}$ can then be removed:
      AUTH$_\tau \leftarrow g$ (AUTH$_{\tau-1}$ || KEEP$_{\tau-1}$), remove KEEP$_{\tau-1}$
   b) The authentication path for leaf $s + 1$ requires new right nodes on heights
      $h = 0, \ldots, \tau - 1$. For $h < H - K$ these nodes are stored in TREEHASH$_h$
      and for $h \geq H - K$ in RETAIN$_h$:
      for $h = 0$ to $\tau - 1$ do
        if $h < H - K$ then AUTH$_h \leftarrow$ TREEHASH$_h$.pop()
        if $h \geq H - K$ then AUTH$_h \leftarrow$ RETAIN$_h$.pop()
   c) For heights $0, \ldots, \min\{\tau - 1, H - K - 1\}$ the Treehash instances must be
      initialized anew. The Treehash instance on height $h$ is initialized with
      the start index $s + 1 + 3 \cdot 2^h < 2^H$:
      for $h = 0$ to $\min\{\tau - 1, H - K - 1\}$ do
        TREEHASH$_h$.initialize($s + 1 + 3 \cdot 2^h$)
5: Next we spend the budget of $(H - K)/2$ updates on the Treehash instances to prepare upcoming
   authentication nodes:
   repeat $(H - K)/2$ times
   a) We consider only stacks which are initialized and not finished. Let $k$ be
      the index of the Treehash instance whose lowest tail node has the lowest
      height. In case there is more than one such instance we choose the instance
      with the lowest index:
      $k \leftarrow \min\left\{h : \text{TREEHASH}_h.\text{height}() = \min_{j=0,\ldots,H-K-1}\{\text{TREEHASH}_j.\text{height}()\}\right\}$
   b) The Treehash instance with index $k$ receives one update: TREEHASH$_k$.update()
6: The last step is to output the authentication path for leaf $s + 1$: return AUTH$_0, \ldots,$ AUTH$_{H-1}$.
```

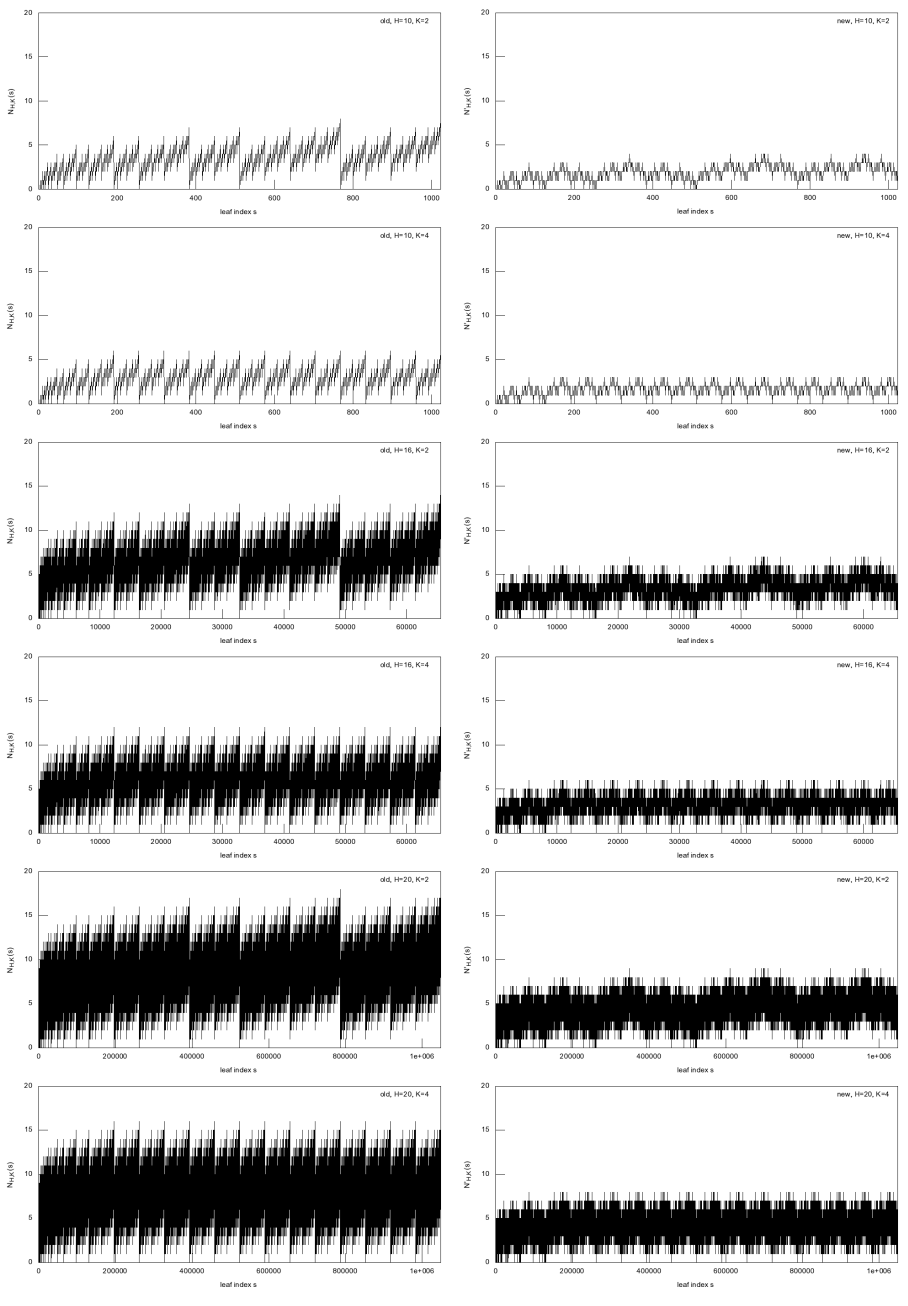

Fig. 3. Comparison of $N_{H,K}\,(s)$ and $N'_{H,K}\,(s)$ for $H = \{10, 16, 20\}$ and $K = \{2, 4\}$ for all leaves s of the respective tree.