AMSA: Adaptive Merkle Signature Architecture

Emanuel Regnath
emanuel.regnath@tum.de
Technical University of Munich, Germany

Sebastian Steinhorst
sebastian.steinhorst@tum.de
Technical University of Munich, Germany

***Abstract*—Hash-based signatures (HBS) are promising candidates for quantum-secure signatures on embedded IoT devices because they only use fast integer math, are well understood, produce small public keys, and offer many design parameters. However, HBS can only sign a limited amount of messages and produce – similar to most post-quantum schemes – large signatures of several kilo bytes.**

In this paper, we explore possibilities to reduce the size of the signatures by 1. improving the Winternitz One-Time Signature with a more efficient encoding and 2. offloading auxiliary data to a gateway.

We show that for similar security and performance, our approach produces 2.6 % smaller signatures in general and up to 17.3 % smaller signatures for the sender compared to the related approaches LMS and XMSS. Furthermore, our open-source implementation allows a wider set of parameters that allows to tailor the scheme to the available resources of an embedded device, which is an important factor to overcome the security challenges in IoT.

***Index Terms*—Hash, Signature, WOTS, IoT**

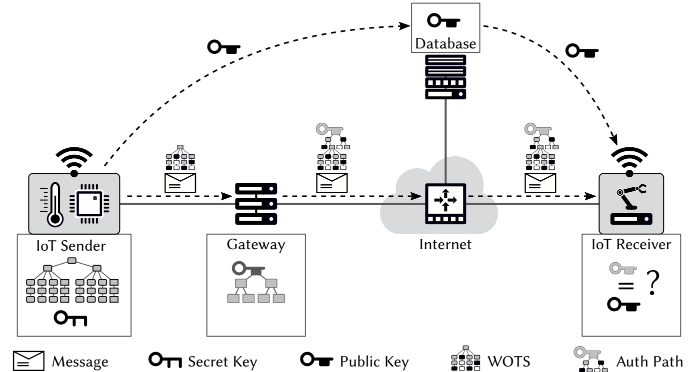

Figure 1: Overview of our signature architecture AMSA. An IoT Sender creates several Winternitz One-Time Signatures (WOTS) from a private key and offloads the authentication path – which connects the WOTS to the public key – to a gateway. Afterwards the IoT sender only needs to create the WOTS to sign a message, reducing the communication overhead for the sender. If a message is sent, the gateway will append the authentication path to complete the signature.

I. Introduction

Security is a difficult design goal in resource constrained IoT environments and, as a result, many vulnerabilities are found in the IoT sector. For authenticated messages, public key cryptography (PKC) has slowly evolved to become usable on embedded devices due to decreased costs for general processing power and hardware accelerators. However, quantum computers with 50 qubits (IBM) and 72 qubits (Google) are already in use [1] and continuously approach the point at which they will be able to completely break currently used PKC such as RSA and ECC [2].

Although 1000 qubits would be required to break 160 bit ECC [2], it is unclear how fast quantum computers will evolve in the near future and, thus, new crypto-systems need to be developed, thoroughly tested, and broadly adopted *before* we reach that threshold. With 24 billion expected IoT devices by the year 2020 [3], we need to find new solutions that are compatible with the resource constraints of these devices.

Hash-Based Signatures (HBS) are promising candidates for quantum-secure signatures on embedded IoT devices. For example, the IETF considers HBS for secure firmware updates of IoT devices [4] and NIST is currently requesting comments for hash-based signature approval [5]. Hash functions are very fast because no floating point operations are required and they are often accelerated in hardware. The security of HBS only relies on the well-studied security properties of the underlying hash function. In most signature schemes, hashing of the message is already required, which would allow to reuse the binary code and build a quantum secure signature based on a single, well-studied cryptographic primitive.

However, HBS produce large signatures and can only sign a limited amount of messages because all signing keys have to be pre-generated before the first use. While the amount of possible signatures can be chosen, the size of the signatures significantly increases the communication overhead, which is a huge challenge in IoT environments where bandwidth is limited.

A. Contributions

We have developed and implemented a lightweight, quantum-secure, stateful Many-Time Signature (MTS) scheme for IoT applications based on a single cryptographic hash function. Our scheme, which is shown in Figure 1, is optimized for highly constrained devices and allows different adaptations to make efficient use of the available resources. In particular, we

- improve the Winternitz One-Time Signature (WOTS) to reduce its size (Section IV-A).
- propose a novel cooperation scheme where a (non-trusted) gateway provides parts of the signature to further reduce the transmission load of the signing IoT device (Section V).
- provide an open-source implementation [6] that can be adapted to specific resource constraints.
- evaluate and compare our approach to the related schemes XMSS and LMS. We show that our scheme is secure considering current and future attack scenarios on cryptographic hash functions.

II. Elements of Hash-Based-Signatures

We will now introduce the core elements and working principles of HBS, which are necessary to understand our

With the support of the Technische Universität München – Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763.

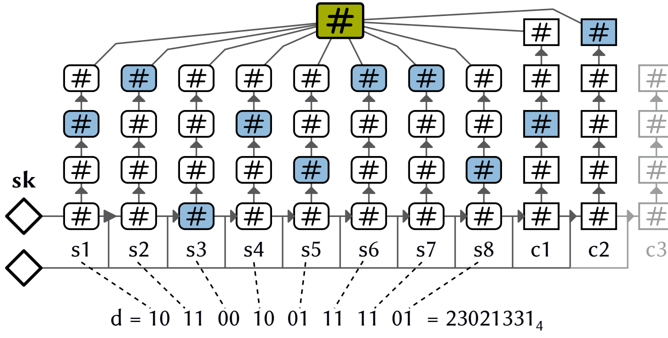

Figure 2: Concept of WOTS illustrated for $n = 2$ and $w = 4$. The digest d is split into equally sized chunks of $\log_2(w) = 2$ bits and each chunk is encoded by one hash-chain s_i. In our variant we allow checksum-chains with different lengths which reduces the total number of chains by one (c_3 not needed).

contributions. Hash-based signature schemes generate key pairs by using cryptographic hash functions. Most schemes first generate several hash-based Few- or One-Time Signatures (OTS) and later combine these with a Merkle Tree to create an MTS, which can be used for a large number of signatures. For an OTS, a random value (private key) is used to generate a set of secrets, which are then individually hashed. All these individual hash digests constitute the public key of the OTS. Since the secrets correspond to the preimages of the hash function, it is infeasible to guess the secrets from the output image (public key). In order to sign a message, a certain set of the secrets is revealed and transmitted together with the message. The receiver then needs to verify that this set of secrets (the signature)

- uniquely encodes the message digest and
- belongs to the public key.

Revealing a unique combination of secrets can only be done once. While some schemes allow to reveal secrets from the same public key a few times, it will always lower its security, and for most schemes (OTS) using them more than once already means broken security.

In order to create an MTS based on OTS, the public keys of several OTS are combined using a Merkle Tree, which is a binary tree of hashes. The root hash of the Merkle tree is then the overall public key of the scheme and each message is signed by an unused OTS at the leafs. This structure is illustrated in Figure 3.

In the remainder of the paper, we assume a single cryptographic hash function $H(\cdot)$ that outputs a hash of n bytes or $N = 8n$ bits:

$$H(\cdot) : \{0,1\}^* \rightarrow \{0,1\}^{8n} \quad (1)$$

A. Lamport OTS

In 1979, L. Lamport proposed the first Hash-based One-Time-Signature (OTS) by using $2N$ hashed secrets in pairs of two in order to encode a message digest of N bits [7]. The hashes of $2N$ secrets are distributed as public key and each pair of secrets is used to encode one bit of the message digest d. Depending on the bit value, one of the two secrets is revealed in the signature while the other one is kept secret.

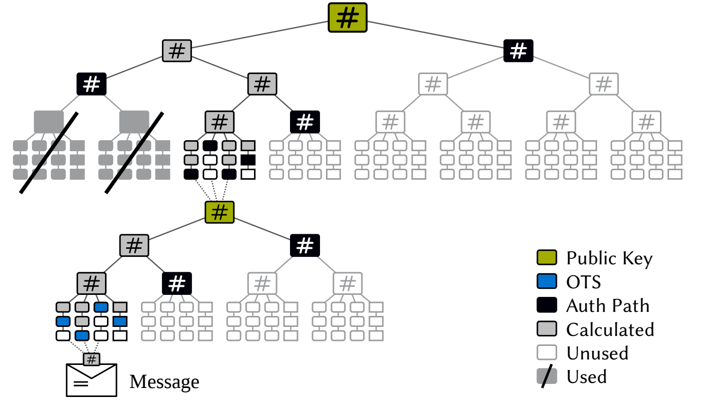

Figure 3: Hash-based MTS with two levels. The hash of the message is signed by a Merkle signature, whose root hash is again signed by a second Merkle signature.

B. Winternitz One-Time Signature (WOTS)

The core idea of the WOTS [8] is to sign multiple bits of the message digest using only one secret of the private key. This is achieved by iteratively hashing each secret w times instead of only once, resulting in several chains of hashes. For signing, we group $\log_2(w)$ bits of the message digest d together, where w is the so called Winternitz parameter. The grouped bits then encode the position of a hash within the chain that will be revealed.

Since revealing any hash within a chain also indirectly reveals all the succeeding hashes in the chain, an attacker could easily forge a signature. For messages where the digest bits are larger or equal to the original message in every group, the attacker could simply reveal a hash at an increased position in the hash chain. In order to prevent this attack, a checksum C is appended to the message that encodes the sum of all indirectly revealed hashes, ensuring that an attacker who wants to increase digest bits must also decrease bits of the checksum C at the same time. This is practically infeasible since in either case the attacker would now need to find the preimage of a given hash.

The checksum is calculated as

$$C = \sum_{i=1}^{\ell_1} ((w-1) - \mathrm{BASE}_w(d_i)) \quad (2)$$

where d_i is the i-th group of w bits of the message digest, w the Winternitz parameter and ℓ_1 the number of hash chains needed to encode the message digest d.

In total, a WOTS requires $\ell = \ell_1 + \ell_2$ separate hash chains of length w. The first ℓ_1 chains are used to encode the message digest d and the last ℓ_2 chains are used to encode the checksum. Both values depend on the chosen w and are calculated as

$$\ell_1 = \left\lceil \frac{8n}{\log_2(w)} \right\rceil \qquad \ell_2 = \left\lceil \frac{\log_2(\ell_1(w-1))}{\log_2(w)} \right\rceil \quad (3)$$

The public key consists of a single hash, which is calculated as the hash of the last hashes of all hash chains concatenated together. Note that the related approach XMSS [9] uses a binary tree instead of concatenation to obtain a single hash.

Figure 2 illustrates an example WOTS where we use $n = 16$ and $w = 4$, which results in $\ell_1 = 8$ and $\ell_2 = 3$. With such an (insecure!) scheme we could sign each $\log_2(w) = 2$ bits of

n	w	ℓ_1	ℓ_2	$\lvert\sigma_W\rvert$	#H
10 B	4	40	4	440 B	176
10 B	16	20	3	230 B	368
10 B	32	16	2	180 B	576
16 B	4	64	4	1088 B	272
16 B	16	32	3	560 B	560
16 B	256	16	2	288 B	4608
20 B	4	80	4	1680 B	336
20 B	16	40	3	860 B	688
20 B	32	32	2	680 B	1088
32 B	4	128	5	4256 B	532
32 B	16	64	3	2144 B	1072
32 B	256	32	2	1088 B	8704
40 B	4	160	5	6600 B	660
40 B	16	80	3	3320 B	1328
40 B	32	64	3	2680 B	2144

n	h	|auth|	|leafs|
10 B	8	80 B	3 kB
10 B	10	100 B	10 kB
10 B	16	160 B	655 kB
16 B	8	128 B	4 kB
16 B	10	160 B	16 kB
16 B	16	256 B	1049 kB
20 B	8	160 B	5 kB
20 B	10	200 B	20 kB
20 B	16	320 B	1311 kB
32 B	8	256 B	8 kB
32 B	10	320 B	33 kB
32 B	16	512 B	2097 kB
40 B	8	320 B	10 kB
40 B	10	400 B	41 kB
40 B	16	640 B	2621 kB

Table I: Typical parameters for WOTS (left) and Merkle trees (right) depending on the underlying hash size n. $|\sigma_W|$ is the size of the WOTS and #H the number of hash operations to generate it.

the message digest d by one chain. The maximum checksum would be $C_{\max} = (4-1)\cdot 8 = 24$ and so we need $\ell_2 = 3$ chains c_i for the checksum. In our example, the checksum is $C = 24 - (2+3+0+2+1+3+3+1) = 24 - 15 = 021_4$. We will explain in Section IV-A how we can reduce ℓ_2 to 2.

C. WOTS+

One problem with WOTS is that the security of the OTS is lower than the security of the used hash function because an attacker just needs to find a preimage for *any* known hash to change the signature. This reduces the number of expected trials until a preimage is found from 2^n to $2^{n-\ell}$.

WOTS+ [10] mitigates the problem by using random values $\vec{r}$ that are XORed with every intermediate hash *before* hashing it again. XORing individualizes the hash calls and ensures that each trial in a brute-force attack is only valid for a single target hash. However, the random values are part of the public key, increasing its size by $n(w-1)$ bits, which is w times larger compared to the conventional WOTS.

D. Merkle Tree

A Merkle Tree is a binary tree entirely made of hash values. Every node corresponds to the hash value of the concatenation of its two child hashes: $h = H(h_{\text{left}}||h_{\text{right}})$. The Merkle tree is used to link several WOTS together and create a single public key which is the root hash of the Merkle tree.

As the name suggests, a WOTS can only be used to sign a single message because the signature contains parts of the secret key. In order to sign several messages with one key pair, the secret key is used as a seed to generate l distinct WOTS. The l public hashes of these l WOTS are then used as leaf hashes to build a Merkle tree of height $h = \lceil \log_2(l) \rceil$.

III. Related Work

There are two other important implementations of stateful hash-based signature schemes (LMS and XMSS) which we will describe in this section. We modified their code to measure the number of hash calls.

Name	GitHub Repository	Version
LMS [11]	github.com/cisco/hash-sigs	d2db1b2
XMSS [9]	github.com/joostrijneveld/xmss-reference	bb2d285

A. Leighton-Micali Signature (LMS)

LMS is a stateful HBS and actively developed as RFC 8554 [11]. LMS uses a security string that is prepended to the input of every hash invocation to mitigate preimage attacks when multiple images of the same hash function are known. The security string is distinct for every hash invocation within and between signature trees, such that any given hash image needs to be attacked with its individual security string.

The security string is up to $21+n$ bytes long and consists of 6 parameters $(I, [r|q], D, [\varepsilon|j|C])$ [11]. I is a random $16\,\mathrm{B}$ identifier for the key-pair, r or q are the $4\,\mathrm{B}$ index of either the node in an authentication path call or the leaf index in an OTS hash call, D is a $2\,\mathrm{B}$ identifier for the context in which the hash function is invoked, j is a $1\,\mathrm{B}$ iteration number for the private key, and C is a n byte random number only used when the message is hashed.

LMS also supports a hierarchical tree structure with several layers of LMS subtrees.

The keys are stored as $privkey = (type, I, seed)$ and $pubkey = (type, I, h_{\text{root}})$, where seed and h_{root} are both n byte and type is four byte. The signature consists of $sig_i = (i, \sigma_i, type, \text{auth}_i)$ for the i-th leaf.

The adjustable parameters are $w \in \{2, 4, 16, 256\}$ and $h \in \{5, 10, 15, 20, 25\}$. The only hash function for all combinations is SHA-256.

B. XMSS and XMSS^{MT}

The eXtended Merkle Signature Scheme [9] consists of a binary hash tree of height h and the 2^h leaf hashes are the root hashes of WOTS+. In the multi-tree variant XMSS^{MT}, several layers of trees are used to increase the total number of signatures.

In contrast to LMS, XMSS only requires a second-preimage resistant hash function because it uses additional bit masks to enhance security. Instead of a security string, each node of the tree is XOR-ed with random bit masks.

The signature therefore contains the 32-bit leaf index i, the n-byte random seed r for the masks, a WOTS+ signature, and an authentication path, summing up to $|\sigma| = (4 + n + (\ell + h)\cdot n)$ byte [9]. The public key consists of typestring, Merkle root, and the seed r, so $(4+n+n)$ bytes.

While the XOR masks allow to prove security in the standard model, they also increase the amount of data needed to verify the validity of a signature and therefore contradict our goal of an IoT-suitable solution.

Furthermore, XMSS uses so called L-Trees to calculate the root hash of each WOTS+, which increases the number of hash operations compared to concatenating all top hashes in one call.

C. Further Signature Schemes

There are also signature ideas around the Distributed Ledger Technology (DLT), such as IOTA Signature Scheme (ISS) [12] and Blockchained Post-Quantum Signature (BPQS) [13].

ISS is IOTA's own variant of the WOTS and removes the checksum chains of the WOTS by balancing the digest hash chains to the expected (mean) checksum value. In the current implementation, ISS simply increases or decreases the message digest until the checksum is balanced, which definitely lowers the security. While the community discusses Proof-of-Work to

find a balanced WOTS, the overall idea is not mature and not scientifically documented.

BPQS is a stateful HBS that uses Blockchain to record the states. Each state consists of a small 2-leaf Merkle tree with two one-time signatures, one for signing a message and the other one for signing the root of the next state. This method allows to sign an infinite amount of messages because the Blockchain keeps track of the current state, such that only the current state needs to be authenticated by the signature.

However, ISS and BPQS require all network participants to keep track of the ledger state in order to verify a signature. This introduces a high memory and communication overhead and thus renders these approaches unsuitable for highly constrained devices. As a result, we will follow their development but not include them in our evaluation.

Furthermore, stateless schemes such as SPHINCS$^+$ [14] are not included in this paper, because they produce much larger signatures ($\approx 40 \pm 10\,\mathrm{kB}$).

IV. Our Adaptive Merkle Signature Architecture

We propose a new MTS, which we call AMSA, based on the Winternitz One-Time Signature (WOTS). One of the main differences is a reduced size of the signature and more parameter choices. Our implementation is published on GitHub [6] for review and further research. This section will describe our optimization mechanisms and implementation details.

A. Our improved MinWOTS

We now introduce our efficient variant called MinWOTS, which reduces the signature size while retaining the full security of the original WOTS. Basically, we reduce the signature size by using a separate and higher w_c for the checksum bits, which is based on the idea discussed in [15].

A typical parameter choice for WOTS is $n = 16, w = 16$ which results in $\ell_1 = 32, \ell_2 = 3$ for the conventional WOTS. Here, we can encode $w^{\ell_2} = 4096$ checksum values. However, the maximum possible checksum value is $w \cdot \ell_1 = 512$, which means that 3584 encodings are not used. Choosing $\ell_2 = 2$ does not work as it can only encode $w^2 = 256$ values. We therefore allow a different hash-chain length w_c only for the checksum bits to encode these bits more efficiently. In our example we would choose $w_c = 23$ to encode up to $23^2 = 529$ values, leaving only $529 - 512 = 17$ encodings unused. More general, w_c is calculated as

$$w_c = \left\lceil \sqrt[\ell_2 - 1]{\ell_1 \cdot w} \right\rceil \tag{4}$$

with ℓ_2 from the conventional WOTS as stated in Equation 3. The found w_c allows us to reduce the number of required hash chains for the checksum by one, meaning $\ell_2' = \ell_2 - 1$. For many usable parameter sets this will reduce $\ell_2 = 3$ to $\ell_2' = 2$, saving n bytes of signature size.

Note that the maximum amount of hash operations for the checksum also decreases from $3w = 48$ to $2w_c = 46$, since the full chain needs to be hashed during signing and verification.

a) Full Rootkey Hashing Another improvement we adopt from [11] is to generate the WOTS-rootkey by hashing all WOTS-pubkeys at once, without using a tree. This reduces the amount of hash operations from $n \cdot (2\ell - 2)$ to $n \cdot \ell$ bytes. For $n = 10, \ell = 23$ this would reduce hashed bytes from 440 to 230.

B. Security String

In order to avoid lowering the security for parallelized brute-force attacks, we use a security string to individualize hash calls. However, we simplify and reduce our security string to a fixed size of 16 random bytes I, which is similar to the identifier I from LMS.

Therefore, we adjust the first 5 bytes of I by setting them to index values in the following way:

- $I[0]$: the WOTS chain index $i_c \in 0..\ell - 1$
- $I[1]$: the hash index within a WOTS chain $i_h \in 0..w - 1$
- $I[2..4]$: the Merkle node index $i_m \in 0..2^{h+1}$

The remaining 11 bytes are unique for the entire key pair and avoid attacks on several keys at once. When calculating the message hash, WOTS public key, or during Merkle hashing, $I[0] = I[1] = 255$.

This way, we ensure that no two revealed hash values, for which the preimage is unknown, can be targeted by the same preimage guess. Note that for each WOTS leaf, the hash calls during WOTS chaining, WOTS public key generation, and Merkle traversing require already different preimages by design since they have different input lengths: n, $n \cdot \ell$, and $2n$.

C. Typecode

The specific parameter choices for n, w, and h are stored in a typecode, which will be part of the public key. While XMSS and LMS use 4 bytes, we use a 2 byte encoding.

Note that we allow any height h between 4 and 20 to enable a better optimization of the scheme to the available resources. In contrast for LMS, where $h \in \{5, 10, 15, 20, 25\}$, choosing $h = 15$ could be already too large for embedded devices, while $h = 10$ only allows to sign 1024 messages.

D. Multi-Layer Merkle Tree

We construct the Merkle Tree similar to LMS and XMSS and allow several layers of trees.

Each Merkle tree in each layer, has its own security string and typecode. In contrast to XMSS, where each tree in each layer needs to have the same typecode, LMS allows different typecode between layers but not between trees. AMSA goes even further and allows choosing any valid parameter set for each tree.

While several layers will increase the signature size, they will significantly reduce the keypair generation time and allow to offload a larger fraction of the signature to a gateway, which will be discussed in the next section.

E. Trade-off: Private Key Compression

There are several possibilities which data is stored as the private key. The fastest signing process can be achieved by storing all hashes of all OTSs and all Tree hashes as the private key. During signing, all required hashes can be picked from memory without any re-computation. While for $(n, w, h) =$

$(32, 16, 10)$ this would mean storing $35\,\text{MB}$ as private key, for $(n, w, h) = (32, 256, 16)$ it scales to $18\,\text{GB}$.

On the other side of the spectrum, only the initial seed for generating all WOTS leafs could be stored but then the entire tree needs to be recomputed for each signature [11].

Therefore, we use our own variant of the Merkle tree traversal algorithm [16], which caches all right nodes in the Merkle tree and only h left nodes. While the left nodes of the tree are calculated when traversing the leafs for each new signature, right nodes are the computationally most expensive nodes to recompute.

The caching is achieved by storing all right nodes (odd index starting from 0) of each level down to level $h - 1$ and all h leftmost nodes during the generation of the key pair.

Whenever we use our key to sign a message, we only need to recompute a single WOTS root (leaf hash on level h). The other leaf hash will be cached from the previous round, such that we always know the first hash of the authentication path. For example, if we sign a left leaf, we will recompute the right leaf hash. If we sign a right leaf, we have cached the left leaf hash from the previous signature.

The remaining hashes of the authentication path can be directly read from the cached tree hashes. Only if a left subtree is exhausted, the cache of left hashes needs to be updated with the root of that subtree.

In total, we cache $2^{h-1} - 1$ right node hashes, h left node hashes and 2 WOTS private keys summing up to $n\left(h + 2^{h-1} - 1\right) + 2n\ell$ bytes.

V. Auxiliary Authentication Gateway

We now describe our idea to reduce the effective signature size by offloading the authentication path of a signature to a gateway.

In a typical IoT scenario, which is shown in Figure 1, we assume a resource-constrained IoT node, which signs messages using an AMSA key-pair and sends them to a receiving node via a more powerful gateway.

This gateway could now be utilized to provide the authentication paths of the AMSA signature. This would release an IoT node from the burden of transmitting the full authentication path of the WOTS each time it signs a message.

After generating the AMSA tree, the IoT node sends all leaf hashes to the gateway. Note that the gateway can not use the leaf hashes to create signatures and therefore there is no trusted relationship between node and gateway.

When the node signs a message using a WOTS, it only sends the WOTS and the leaf index to the gateway. The gateway constructs and appends the authentication path to the message before forwarding the message to the receiver. Offloading the authentication path has several advantages:

1) The effective signature size for the IoT node is reduced.
2) The IoT node does not need to construct the auth-path and thus only needs to store the current WOTS and the next seed.
3) In a multi layer signature, all parts but the bottom most WOTS could be offloaded.

Algorithm	XMSS	LMS	AMSA	ECC
|PK|	68 B	56 B	50 B	64 B
$\|\text{SK}_{\text{min}}\|$	64 B	48 B	50 B	32 B
|Sig.|	2500 B	2504 B	* 2434 B	64 B
$\#H_{\text{gen}}$	1166345	1098761	1082377	N/A
$\#H_{\text{sign}}$	579	512	521	N/A
$\#H_{\text{verify}}$	613	546	554	N/A

Table II: Theoretical comparison of stateful HBS schemes for $n = 32, h = 10, w = 16$ and ECC for $n = 32$ as reference. While all HBS have similar key sizes and number of hashed calls, AMSA provides the smallest signature. *: For normal operation. If the auxiliary gateway is used, the signature size for the sender is 2146 B (11.8 % reduction).

a) Example If the gateway constructs the authentication paths, then the IoT node needs to transmit 2^h leaf hashes during key generation and afterwards only the OTS. The effective signature size for the node over all 2^h signatures is:

$$\frac{n \cdot 2^h + n \cdot 2^h \cdot \frac{8n^h}{\log_2(w)}}{2^h} = n + n \cdot \frac{8n^h}{\log_2(w)} \tag{5}$$

Basically, the transmission overhead for the authentication paths per signature is reduced from $h \cdot n$ to only n. For $h = 10, n = 20, w = 16$ this would result in $(840 + 20) = 860\,\text{B}$ instead of $(840 + 200) = 1040\,\text{B}$, which is a signature reduction by 17.3%. With a higher n or lower h, this percentage becomes smaller, which is why we state 17.3 % as the possible reduction.

The efficiency for multi layer signatures is almost the same. In case we use two layers with $h_0 = 10, h_1 = 10$ to get the same number of signatures for the first bottom tree, we need to also transmit the WOTS and the auth. path of the top tree which would be additional $1040\,\text{B}$ over 1024 signatures. In summary, this would result in $(840 + 20 + \frac{1040}{1024}) = 861.02\,\text{B}$ per signature.

Overall, utilizing the gateway for providing the authentication path, can significantly reduce the computational effort and transmitted data of the signed messages for the IoT nodes connected to the gateway.

VI. Analysis

In this section we will evaluate our implementation, which is written in C, and compare it to related implementations regarding key and signature sizes, computational performance, binary size, and code size. The results are summarized in Table II and Table III.

A. Performance

The speed of the scheme clearly depends on the number of hash operations. From analyzing the call graph, we found that 94% of the CPU time of the `AMSA_sign` function is spend in the hash compression function. This means that a huge performance improvement is possible if the hash function is hardware accelerated.

The hash function is called with four different input lengths: 1) the length of the message, 2) n for generating WOTS chains, 3) $n \cdot \ell$ when calculating the WOTS root, 4) $2n$ when calculating Merkle nodes. Since the execution time is proportional to the input length, we compare the approaches based on the total amount of data x that is hashed.

Algorithm	XMSS	LMS	AMSA	uECC
$\lvert x_{\text{gen}}\rvert$	331.6 MB	60.8 MB	37.8 MB	N/A
$\lvert x_{\text{sign}}\rvert$	468.4 kB	152.1 kB	55.3 kB	N/A
$\lvert x_{\text{verify}}\rvert$	182.9 kB	27.3 kB	19.1 kB	N/A
t_{sign} on Intel i7	2.2 ms	1.3 ms	0.79 ms	0.61 ms
t_{verify} on Intel i7	0.81 ms	0.31 ms	0.24 ms	0.69 ms
t_{sign} on Cortex M0	3004 ms	1373 ms	431 ms	841 ms
t_{verify} on Cortex M0	808 ms	143 ms	152 ms	438 ms
|Binary|	151.1 kB	107.3 kB	34.5 kB	36.6 kB
LOC	1.9 k	3.9 k	1.1 k	33.6 k

Table III: Experimental comparison of stateful HBS schemes and ECC as a reference. We state computational effort as the total amount of input $|x|$ (in bytes) to the hash function H and as specific timings t. We compiled each HBS for `SHA256_W16_H10` and uECC [17] for `secp256r1` using `-O3`. The LOC were counted using the tool `cloc` and skipped implementations of hash functions.

VII. Security

The security of a signature is based on the difficulty for an attacker to forge a valid signature/message pair. For HBS, this difficulty relies on the security of the underlying hash function, which is discussed by three resistances:

1) First-preimage: difficulty to find a preimage x of one known image $h = H(x)$.
2) Second-preimage: difficulty to find a second preimage y with $H(y) = h$ of one known preimage-image pair x, h with $h = H(x)$.
3) Collision: difficulty to find any two values a, b that result in the same image $H(a) = H(b)$.

A. Preimage Attacks

Hash-based Signatures can be forged if an attacker can find a preimage for one of the revealed hash values. However, not all hash values are equally important.

For example, if an attacker can find the preimage of a hash in the WOTS chain, he/she can change the signature by only a single bit. If the checksum chain was attacked, the attacker can change ℓ_1 bits. In contrast, another preimage for the message hash or the WOTS Root hash would allow to sign a completely different message. Attacking the Merkle tree is most profitable, because in the case of success, an attacker could forge up to 2^{h-1} arbitrary messages. To do so, the attacker would construct a new $h - 1$ AMSA tree with a public key PK' and then tries to find any n-byte value x such that $PK = H(PK'||x)$ completing the tree to its full height.

However, already for a 128 bit hash, it is very difficult to find another preimage. Even if we assume the *entire* Bitcoin network with a current hash rate of $\approx 80 \times 10^{18}$ hashes/s focused on one 128 bit hash preimage attack, it would still take an expected time of $2^{127}/80 \times 10^{18} = 67$ billion years.

B. Preimage attacks on quantum computers

The overall performance of a quantum computer relies on several factors including number of qubits, coherence time, and error rates [2] and has been continuously growing over the last years.

While the security of ECDSA and RSA would be completely broken by Shor's algorithm, the security of hash functions is only reduced to half by Grover's algorithm.

However, results from [18] suggest that a real QC preimage attack of SHA-3-256 would require $\approx 2^{166}$ operations instead of the theoretical optimum of 2^{128}. In general, current research suggests that hash functions provide at least the same security against QC attacks compared to classical attacks when their number of digest bits is doubled. Therefore, choosing $n = 256$ for HBS provides at least 128 bit security against quantum computers.

VIII. Conclusion

Quantum secure schemes increase the size of signatures and keys compared to classical schemes such as ECC. Existing HBS implementations only allow a very narrow set of parameters, which diminishes the flexibility of HBS.

Our AMSA leverages the large variety of parameters to enable adaption of the scheme to the available resources of devices. This adaption is crucial to overcome the security challenges for constrained devices in an efficient manner.

When identical security parameters are compared to state-of-the-art HBS, AMSA provides 2.6 % smaller signatures in general and 17.3 % smaller signatures for the sender if an auxiliary gateway is used.

References

[1] B. Villalonga *et al.*, "Establishing the quantum supremacy frontier with a 281 pflop/s simulation," *arXiv preprint arXiv:1905.00444*, 2019.

[2] V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang, "The Impact of Quantum Computing on Present Cryptography," *arXiv preprint arXiv:1804.00200*, 2018.

[3] A. R. Sfar, E. Natalizio, Y. Challal, and Z. Chtourou, "A roadmap for security challenges in the Internet of Things," *Digital Communications and Networks*, vol. 4, no. 2, pp. 118 – 137, 2018.

[4] B. Moran, M. Meriac, H. Tschofenig, and D. Brown, "A Firmware Update Architecture for Internet of Things Devices," Internet Engineering Task Force, Internet-Draft, Apr. 2019.

[5] NIST Computer Security Resource Center, "Request for Public Comments on Stateful Hash-Based Signatures," 02 2019.

[6] "AMSA Code," https://github.com/tum-esi/AMSA.

[7] L. Lamport, "Constructing digital signatures from a one-way function," CSL-98, SRI International Palo Alto, Tech. Rep., 10 1979.

[8] R. C. Merkle, "A certified digital signature," in *Proceedings on Advances in Cryptology*, ser. CRYPTO '89. New York, NY, USA: Springer-Verlag New York, Inc., 1989, pp. 218–238.

[9] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme," IRTF, RFC 8391, 5 2018.

[10] A. Hülsing, "Wots+ – shorter signatures for hash-based signature schemes," *Cryptology ePrint Archive*, no. 965, 2017.

[11] D. McGrew, M. Curcio, and S. Fluhrer, "Leighton-Micali Hash-Based Signatures," IRTF, RFC 8554, 4 2019.

[12] W. Pinckaers, "Iota signatures, private keys and address reuse?" Blog Article, 3 2018, http://blog.lekkertech.net/blog/2018/03/07/iota-signatures/.

[13] K. Chalkias *et al.*, "Blockchained post-quantum signatures," in *IEEE iThings and GreenCom and CPSCom and SmartData*. IEEE, July 2018.

[14] D. J. Bernstein *et al.*, "The SPHINCS+ Signature Framework," in *Computer and Communications Security*, 11 2019.

[15] S. Even, O. Goldreich, and S. Micali, "On-line/off-line digital signatures," in *Advances in Cryptology (CRYPTO 89)*. Springer, 1990, pp. 263–275.

[16] J. Buchmann, E. Dahmen, and M. Schneider, "Merkle tree traversal revisited," in *International Workshop on Post-Quantum Cryptography*. Springer, 2008, pp. 63–78.

[17] K. MacKay, "Micro-ECC," https://github.com/kmackay/micro-ecc.

[18] M. Amy *et al.*, "Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3," in *International Conference on Selected Areas in Cryptography*. Springer International Publishing, 2016, pp. 317–337.