Agile Acceleration of Stateful Hash-based Signatures in Hardware

JAN PHILIPP THOMA, DARIUS HARTLIEF, and TIM GÜNEYSU, Ruhr-University Bochum, Germany, and DFKI GmbH, Germany

With the development of large-scale quantum computers, the current landscape of asymmetric cryptographic algorithms will change dramatically. Today's standards like RSA, DSA, and ElGamal will no longer provide sufficient security against quantum attackers and need to be replaced with novel algorithms. In the face of these developments, NIST has already started a standardization process for new Key Encapsulation Mechanisms (KEMs) and Digital Signatures (DSs). Moreover, NIST has recommended the two stateful Hash-Based Signatures (HBSs) schemes XMSS and LMS for use in devices with a long expected lifetime and limited capabilities for maintenance. Both schemes are also standardized by the IETF.

In this work, we present the first agile hardware implementation that supports both LMS and XMSS. Our design can instantiate either LMS, XMSS, or both schemes using a simple configuration setting. Leveraging the vast similarities of the two schemes, the hardware utilization of the agile design increases by 20% in LUTs and only 3% in Flip Flops (FFs) over a standalone XMSS implementation. Furthermore, our approach can easily be configured with an arbitrary number of hash cores and accelerators for the one-time signatures for different application scenarios. We evaluate our implementation on the Xilinx Artix-7 FPGA platform, which is the recommended target for PQC implementations by NIST. We explore potential tradeoffs in the design space and compare our results to previous work in this field.

CCS Concepts: • **Hardware**; • **Security and privacy → Hardware security implementation**;

Additional Key Words and Phrases: XMSS, LMS, hardware implementation, post-quantum cryptography

ACM Reference format:
Jan Philipp Thoma, Darius Hartlief, and Tim Güneysu. 2024. Agile Acceleration of Stateful Hash-based Signatures in Hardware. *ACM Trans. Embedd. Comput. Syst.* 23, 2, Article 29 (March 2024), 29 pages.
https://doi.org/10.1145/3567426

1 INTRODUCTION

Continuing development in the area of quantum computers will lead to a security breach of current asymmetric cryptography schemes in the not-too-distant future. This especially includes the widely deployed RSA and DSA algorithms. In 1999, Peter Shor presented Shor's algorithm, which

The work presented in this article has been partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 39078197 and the German Federal Ministry of Education and Research (BMBF) under the project "QuantumRISC" (ID 16KIS1038) [40] and project "PQC4MED" (ID 16KIS1044).
Authors' address: J. P. Thoma, D. Hartlief, and T. Güneysu, Ruhr-University Bochum, Universitätsstr. 150, Bochum, NRW, Germany, 44801, and DFKI GmbH, Robert-Hooke-Straße 1, Bremen, Bremen, Germany, 28359; emails: {jan.thoma, darius.hartlief, tim.gueneysu}@rub.de.

1539-9087/2024/03-ART29
https://doi.org/10.1145/3567426

solves the factorization and the discrete logarithm problem on a quantum computer in polynomial time [41]. The current understanding of quantum computers and quantum algorithms is that symmetric cryptographic primitives like hash functions are less severely affected. Grover's algorithm [23] roughly halves the security level of these schemes. While Grover's and Shor's algorithms have been known for many years, only the recent advances in the scale and stability of quantum computers [2] have led to a demand of standardization for post-quantum cryptography. Therefore, NIST has started a multi-year competition to find **Key Encapsulation Mechanisms (KEMs)** and **Digital Signature (DS)** algorithms to be standardized [37]. In this competition, a variety of schemes based on codes [6], lattices [4, 14, 17, 22, 45], hashes [3], Isogenies [27], and multivariates [16] has been proposed and evaluated. With the standardization still ongoing, NIST also issued a recommendation for stateful **Hash-Based Signature (HBS)** schemes, namely **Leighton-Micali Signatures (LMSs)** and the **eXtended Merkle Signature Scheme (XMSS)** [15]. NIST recommends these algorithms especially for devices with a long expected lifetime and limited ability to update the algorithm once the device is deployed. The algorithms are also standardized by the IETF in RFC 8391 (XMSS) [25] and RFC 8554 (LMS) [34].

The two stateful HBSs are considered very conservative candidates regarding their security level, that is, since the security of hash functions on which the security of XMSS and LMS is built is well understood and has been extensively studied for decades. A further advantage of the schemes is that many consumer- and server-grade devices feature hardware-accelerated hash cores that can be leveraged to accelerate the schemes significantly. However, many embedded devices do not implement ISA extensions for hash acceleration. Even if a hardware-accelerated hash core is available on the target, this is often not sufficient to meet the strict timing constraints. Our implementation can utilize many parallel hash cores and, hence, accelerate the HBS more efficiently. Embedded devices often perform safety-critical operations and are deeply embedded in, e.g., industrial control units or cars. Hence, they match all the criteria of the NIST recommendation and are therefore an important target for optimized implementations. With sharp timing constraints and low-end CPUs, native software execution of the schemes is often not an option. Due to the safety-critical role that embedded devices play in many applications, certification and verification of crucial building blocks such as the implementation of digital signature schemes are very common in the embedded industry. A full hardware implementation of the signature scheme is advantageous in this case compared to a hardware/software co-design [36, 46] since the complexity of the verification and certification is reduced to the hardware only, and the certified component can be reused in many different device types. This reusability compensates for the initial higher complexity of developing a full hardware design. Moreover, by avoiding a communication bottleneck between the software and hardware layer, it is possible to accelerate the schemes more efficiently. This especially holds if **Direct Memory Access (DMA)** is available. Hence, having a dedicated hardware accelerator for the signature scheme that frees up CPU time and accelerates the signature algorithm can be crucial on embedded devices.

In this work, we implement such an efficient and agile combined hardware accelerator for XMSS and LMS. We present the first agile hardware implementation that supports both stateful HBS schemes. Using a configuration parameter, the design either implements a standalone LMS, a standalone XMSS, or an agile design of both that leverages the similarities of the schemes to reduce the area overhead. Instantiating a single scheme has the advantage of reduced area utilization, while the agile version supporting both schemes is much smaller than two separate accelerators for both schemes. Our design is highly configurable regarding many parameters defined in the RFCs as well as performance tradeoffs like the number of used hash cores. We present the exact configuration parameters in Section 4.1. For the parallel hash cores we use a dynamic scheduling approach for the assignment and separate the hash accelerator from the actual signature accelerator using a bus

system. The advantages of this separation are twofold: First, the implementation of the hash core can easily be exchanged without needing to change the implementation of the signature itself. Second, the interface to the hash accelerator can be made available to the outside to accelerate generic hash operations outside of the signature scheme.

In summary, our work will provide following contributions:

- We present the first full hardware implementation of LMS, combined with an agile hardware implementation of the two stateful HBS schemes, LMS and XMSS. The similarities of the two schemes allow for an efficient, area-optimized design.
- The implementation is highly configurable, including the number of instantiated hash cores and the parameters of the stateful HBS schemes.
- We explore the design tradeoffs and performance differences of XMSS and LMS and compare them to each other.

1.1 Organization of the Article

In Section 2 we introduce related work in the field of hardware acceleration of PQC schemes. Section 3 defines the notation used in this article and introduces the schemes XMSS and LMS. In Section 4, we describe the most important aspects of our hardware implementation. We evaluate the design and compare it to related work in Section 5. Finally, we conclude in Section 6.

2 RELATED WORK

XMSS was introduced in 2011 as a quantum-secure digital signature scheme with minimal security assumptions in [9] and has since been standardized by the IETF in RFC 8391 [25]. The scheme is closely related to the **Leighton-Micali Signature (LMS),** which is also standardized by the IETF [34]. LMS has been proven secure in the quantum random-oracle model in [19]. A comparison between the two schemes has been conducted in [11, 28]. In [39], the WOTS scheme on which both XMSS and LMS built is optimized for either fast verification or fast signature generation. The side-channel security of XMSS has been studied in [29], which was improved upon in [47]. A hardware/software co-design of XMSS using the RISC-V architecture was implemented in [46]. A pipelined ASIC accelerator for the leaf node generation of XMSS was presented in [36]. We implemented the first full hardware accelerator for XMSS in [44]. This work builds on our previous experience with the standalone XMSS implementation and extends it with support for the LMS signature scheme. Furthermore, this work implements the BDS algorithm for more efficient tree-hashing, especially for large tree heights. A different full hardware implementation of XMSS has been presented in [12]. A hardware accelerator for the LMS key generation has been implemented in [42]. Their design targets Xilinx UltraScale FPGA series and is suited to accelerate the key generation for general-purpose CPUs. Our implementation targets embedded use cases and therefore does not compare in the achieved performance but also uses significantly fewer area resources.

There are several other proposals for quantum-secure signature schemes, most of which are part of the NIST **post-quantum cryptography (PQC)** competition [37]. In [38], a hardware accelerator for the number theoretic transform is implemented using high-level synthesis. This accelerator can be utilized to speed up quantum-secure signature schemes such as Falcon [22] and Crystals-Dilithium [18], the latter also being implemented directly in hardware in [33]. In [5], an agile crypto co-processor is implemented that can accelerate various lattice-based schemes. The multivariate signature scheme Rainbow[1] was presented in [16] and implemented in hardware in [21].

[1]Recent attacks put the security of Rainbow in doubt [7].

Table 1. Mapping of Symbols Used in This Work to the Symbols Used in the Respective RFC Documents of XMSS [25] and LMS [34]

Symbol	LMS	XMSS	Description
n	n	n	Hash digest length in bytes
w	2^w	w	Winternitz parameter
ℓ	p	len	Winternitz chains per OTS
h	h	h	Merkle Tree height
k	-	-	BDS parameter

3 BACKGROUND

In this section we focus on the design of the stateful hash-based signature schemes LMS [34] and XMSS [9]. The two schemes feature extensive similarities in their design, which have been analyzed in [11, 28]. Both schemes build on a Merkle Tree construction [35] and use an **One-Time Signature (OTS)** scheme based on Merkle's Winternitz one-time signatures [35]. Despite the structural similarities of the schemes, XMSS takes weaker security assumptions by not relying on the random oracle model for the security proof [9]. LMS, on the other hand, has only been proven secure in the random oracle model [30].

In the following, we lay out the workings of the key and signature generation and the verification algorithm of the two schemes. Finally, we summarize the BDS algorithm, which gives a tradeoff between the latency of the signature generation and the memory used for caching intermediate values of the tree construction.

3.1 Notation

The RFCs for LMS and XMSS use different notations for the same parameters. Table 1 lists the mapping of the notation used for the remainder of the article to the respective standard.

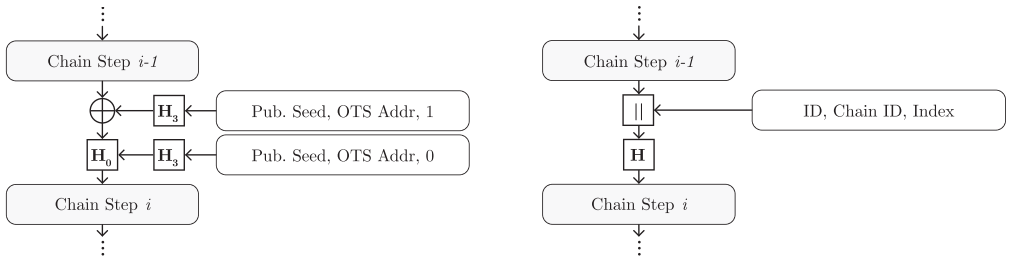
3.2 One-time Signatures

Both LMS and XMSS use variants of the **Winternitz One-Time Signature (WOTS)** scheme, which builds digital signatures from a secure hash function. The WOTS scheme is parameterized by the Winternitz parameter w. From a high-level perspective the scheme works by splitting an n-byte message digest into $log_2(w)$ bit-sized blocks and signing them each using a *hash chain*. In addition to the $\ell_1 = \lceil n/log_2(w) \rceil$ blocks of the message, a checksum is computed over the message and also signed. The checksum consists of $\ell_2 = \lfloor log_2(\ell_1(w-1))/log_2(w) \rfloor + 1$ blocks and computes as

$$c = \sum_{i=1}^{\ell_1} w - 1 - m_i = c_1 || c_2 || ... || c_{l_2}.$$

Hence, in total, the OTS consists of $\ell = \ell_1 + \ell_2$ hash chains. Each hash chain is constructed by first uniformly sampling a secret initial n-byte value from $\{0,1\}^{8n}$ and then applying a one-way function defined by $H : \{0,1\}^* \rightarrow \{0,1\}^{8n}$, $(w-1)$ times to it. The secret key for WOTS consists of the ℓ randomly chosen n-byte words. The WOTS public key contains the result of the hash-chain computation of length $w-1$ for each of the ℓ secret keywords.

To sign a message using WOTS, first the checksum c is computed. The checksum is added to defend against an attacker forging a signature by applying the hash function to any block, creating a valid signature for a message $\bar{m}$ with $\bar{m}_i \geq m_i \forall i \in [1, \ell_1]$. Then, the message and the checksum



(a) WOTS+ used in XMSS (b) LM-OTS used in LMS

Fig. 1. One round of the OTS chaining algorithm used in XMSS (left) and LMS (right). In XMSS the hash function is keyed using a pseudorandom key and operates on the masked output of the previous iteration. LMS simply hashes the concatenated value of the previous iteration value and an identifier.

are concatenated and divided into $log_2(w)$ bit blocks:

$$m = m_1||...||m_{\ell_1}||c_1||...||c_{\ell_2}, m_i, c_i \in \mathbb{F}_2^{log_2(w)}.$$

The one-time signature is generated by iteratively hashing each secret key value s_i, using the respective message block value as the number of iterations to apply. Since each block is $log_2(w)$ bit long, its integer value is between 0 and $w-1$, making the signature some iteration in the hash chain between the secret key and the public key. The signature consists of ℓ hash digests, each representing an intermediate value of the hash chain between the secret key and the public key. Note that the signature may contain parts of the secret key and/or the public key. To verify the signature for a given message, again, first the checksum is computed. Then, each signature chain is continued by hashing it $w-1-m_i$ times, which results in the public key *if* the signature is valid.

The OTS scheme used in LMS is dubbed *LM-OTS* and differs slightly from the original WOTS scheme. For the chain construction, LM-OTS adds an identifier of the current tree, the current chain, and the index within the chain to the hash input in each iteration. The process is shown in Figure 1(b). Instead of using the ℓ n-byte words as public key, LM-OTS compresses the public key to a single hash digest using a one-way function.

XMSS also uses a variation of the WOTS scheme, which is described in [24]. This scheme, called WOTS+, introduces a random mask value in each hash chain iteration, so that each chain iteration operates on a masked input of the previous iteration. Moreover, XMSS uses a keyed hash function with a unique pseudorandom key in every iteration. In XMSS, the bitmasks and keys are generated pseudorandomly from a public seed, which is part of the public key. Therefore, the hash function is used in different modes: The PRF mode (H_3) prepends a zero-padded block with a single "3" to the hash input, while the actual chain step (H_0) prepends a full-zero block to the hash input. One iteration of the WOTS+ chaining algorithm is shown in Figure 1(a). Like LMS, XMSS also compresses the WOTS+ public key to a single hash digest. Instead of simply hashing the entire public key, XMSS uses an unbalanced binary tree construction called L-Tree. The L-Tree is structurally similar to a Merkle Tree (see Section 3.3). Each n-byte word of the public WOTS+ key is assigned to a leaf node of the tree. The leftmost node of a level without an even number of nodes is lifted to the nearest higher level where it can be hashed with another node.

3.3 Merkle Tree Construction

A Merkle Tree Signature uses a balanced binary tree of height h in which each node contains a hash digest (see Figure 2). Every leaf node of the tree contains a compressed public key of a unique OTS. The tree is constructed from the leaves to the root node. On each level, two sibling nodes are

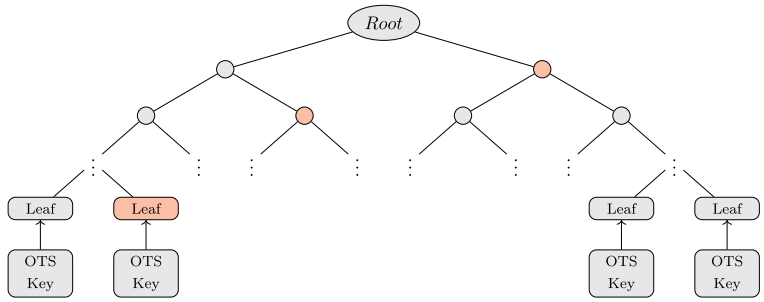
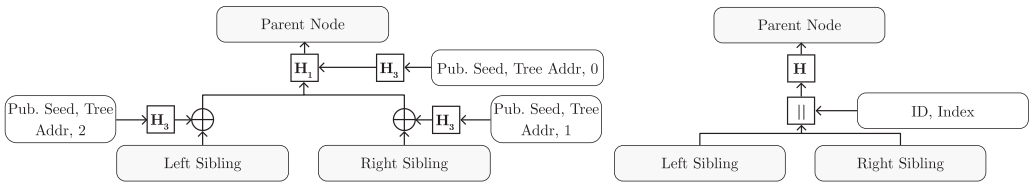


Fig. 2. A generic Merkle Tree signature scheme. The authentication path for the leftmost leaf is highlighted in red.



(a) XMSS Treehash (b) LMS Treehash

Fig. 3. One iteration of the compression used in the treehash algorithm of XMSS (left) and LMS (right). LMS omits the bitmasks and the keyed hash function.

compressed to a new digest that makes up the parent node within the Merkle Tree using a hash function. The root node of the tree is the public key for the signature scheme. Hence, to generate the public key, the complete Merkle Tree must be constructed.

The Merkle Tree constructions of XMSS and LMS are slightly different from each other. In the LMS scheme, each hash input is prepended with a unique prefix to avoid collisions. Otherwise, LMS follows the simple original Merkle Tree construction by hashing the two sibling nodes. The tree construction in XMSS is more complex than for LMS. To generate a root node using XMSS, each node of a sibling pair is first masked using a pseudorandom bitmask and then combined using a keyed hash function with a pseudorandom key. The keyed hash function (H_1) prepends a zero-padded "1" digit to the hash input to distinguish it from hash computations made in the WOTS+ part of the algorithm. Overall, the changes in the treehash function of XMSS are similar to the ones made in WOTS+, which, in combination, make for the improved security properties of XMSS over LMS. The compression of two sibling nodes for both schemes is depicted in Figure 3.

For the key generation, 2^h OTS public keys need to be generated. Using the public keys of the OTS in the leaf nodes of the Merkle Tree, the root node of the tree can be constructed, which serves as the public key for the grand signature scheme. To sign a message, it is first compressed to an n-byte hash digest and then signed using the OTS scheme. The signature consists of the OTS signature and an *authentication path*. The authentication path contains all $h-1$ sibling nodes of the nodes on the path from the current leaf to the root node. The authentication path for the leftmost node is shown in Figure 2. To verify the signature, the verifier first computes the public key of the OTS from the one-time signature and compresses it to a Merkle Tree leaf node. Then, they use the authentication path nodes to calculate the root node of the verification tree. Only if the OTS signature was valid will the calculated public key of the OTS match the actual public key of the OTS, and hence, the calculated root node matches the root node of the Merkle Tree contained in the public key of the grand signature scheme. If this is the case, the signature is valid.

XMSS and LMS are *stateful* signature schemes. Since the leaf nodes are instantiated with OTS, each leaf node can only be used once to sign a message. The state that needs to be maintained internally keeps track of which OTSs have been used already. This is only required for the signer, but not for the verifier. Therefore, the height of the Merkle Tree determines the maximum number of messages that can be signed using the tree. When all leaf nodes have been used, a new public key needs to be generated. Statefulness is the major drawback of stateful hash-based signatures compared to other post-quantum-secure schemes based on, e.g., lattices [4, 14, 17, 22, 45], or codes [6].

3.4 BDS Tree Traversal

Since the signature generation requires intermediate nodes of the Merkle Tree to construct the authentication path, the tree needs to be partially reconstructed for every signed message. While it is possible to store the entire tree during the key generation phase, this results in a large storage overhead: all 2^h leaf nodes and $2^h - 1$ intermediate nodes with n-bytes each would need to be stored. To reduce the memory requirements, one can choose to reconstruct the tree from the seed used for initial key generation. However, this brings a huge performance overhead since especially the computation of the OTS is very complex. Several algorithms providing a tradeoff between these two extremes have been proposed [10, 26, 43]. In this work, we use the BDS algorithm [10], which is the most recent and has a lower worst-case running time compared to the others. Furthermore, it is configurable over the parameter k, which allows a time-memory tradeoff.

Conceptually, a sign operation in the BDS algorithm generates the next authentication path and precomputes nodes needed in the paths of upcoming signatures. The first path is generated during the key generation phase, which allows the computation of the next authentication path during each sign operation. The signing algorithm generates left and right authentication nodes differently. Generally, the computation of left nodes is less expensive than that of right nodes. For left nodes the two child elements are always the previous and current authentication nodes of the preceding height. By keeping track of previous authentication nodes in the *keep* array, the computation of upcoming left authentication nodes only requires a single hash operation that combines two sibling nodes in the Merkle Tree. The generation of right authentication nodes is split further into two methods. Any right node of height h' with $h - k \leq h' \leq h - 2$ is stored in the array $retain_{h'}$ during the key generation phase and loaded from the array into the authentication path during the signature generation. For all heights $h' < h - k$ an instance of a procedural treehash algorithm is used for generating the right nodes of that height. An instance either holds a single finished node value or is currently working on the next right node of that height. The instances first store the second right node of their respective height during key generation as their finished node, because the first right node of every height is part of the initial authentication path. When a finished node has been loaded into the path, the treehash instance is reset and then generates the next right node of its height during the following signing operations using a shared stack in update functions. A function call generates a single leaf and all parent nodes, which can be generated with nodes stored on the stack in previous update procedures. If the height of the instance is not reached, the last node generated is saved on the stack. The BDS algorithm assigns a budget of $\frac{H-K}{2}$ updates to each sign operation [10].

4 IMPLEMENTATION

In this section, we describe the details of our implementation. In the following, we will refer to three different configurations of this implementation. The XMSS- and LMS-only configuration instantiate only XMSS or LMS, respectively. The agile configuration instantiates both schemes in parallel and allows selecting which scheme to be used during runtime. Our implementation is available at https://github.com/Chair-for-Security-Engineering/XMSS-LMS-HW-Agile.

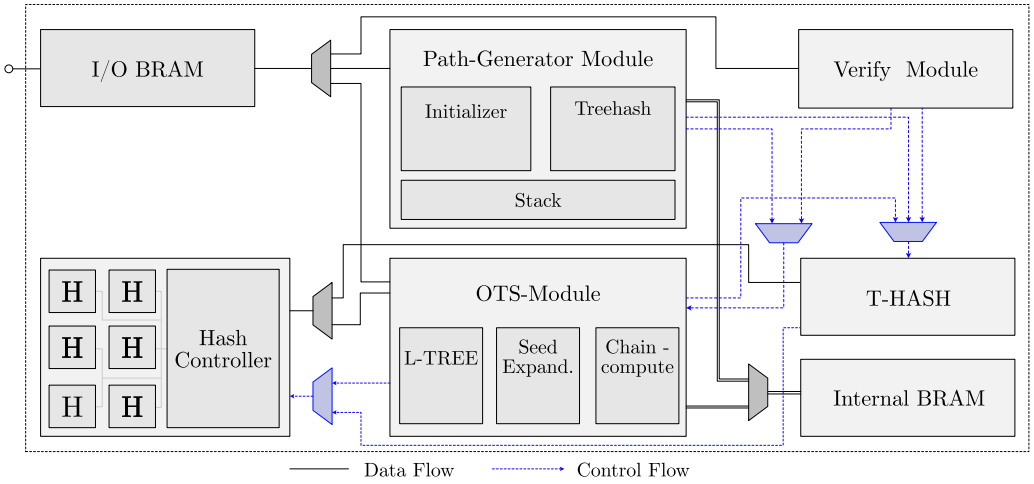


Fig. 4. Block design overview of the architecture.

4.1 Design Rationale

The main design target of our implementation is an agile design that can easily be configured to suit different use cases. This agility ranges from the ability to select the supported algorithm from XMSS or LMS during synthesis *or* runtime, over the number of hash cores used to accelerate the scheme, to parameter setting of the scheme itself like the tree height or the Winternitz parameter. The configuration file is shown in Listing 1. Our design allows selecting the signature scheme, the hash-digest length, the WOTS parameter, the tree height, and performance-related parameters including the number of parallel hash cores instantiated. All parameters can be modified independently. However, since the parameter N is directly influenced by the choice of the hash function, changes may be necessary to support different hash digest lengths. We chose to not implement the multi-tree variants of XMSS and LMS, which would add additional complexity and therefore increase the hardware footprint. The HBS accelerator is implemented fully in hardware, without the need for software support. In addition to the verification and certification benefits mentioned earlier, this can yield performance benefits compared to hardware/software co-design like [36, 46]. Opposed to such designs, the full hardware implementation allows us to optimally schedule parallel computations for a high-capacity utilization of the hash cores. In hardware/software co-designs, the communication between the two layers could easily become a bottleneck, especially if additional tasks are executed on the software level. It must be noted that if the target device does not implement DMA, the hardware accelerator also encounters additional overhead when transmitting the results from the hardware to the software. The magnitude of this delay depends on the device itself. For the multi-tree variants, we believe that a hardware/software co-design utilizing a modified variant of our implementation to compute the subtrees would be well suited, that is, since the computation of the subtrees is very complex by itself and therefore only little communication between the hardware and the software needs to occur. The similarities of XMSS and LMS make it possible to share much of the hardware logic between the two signature schemes. Hence, the hardware overhead for configuring the runtime-agile version should be kept low. Since both XMSS and LMS have been standardized, it is likely that a mixture of systems using XMSS or LMS will evolve in the field. By supporting both schemes, our implementation is compatible with all those systems by different vendors using different standards.

Our design implements the key generation, signature generation, and verification algorithm of XMSS and LMS. Hence, it can be used to fully accelerate both schemes. We use the BDS

algorithm [10] to optimize the runtime of the signature generation while keeping the BRAM utilization low. We target a Xilinx Artix-7 FPGA. A schematic overview of our implementation is shown in Figure 4.

```
package config is
    constant SCHEME: scheme_t := LMS; -- LMS, XMSS or DUAL_SHARED_BRAM

    constant N: integer := 32;
    constant WOTS_W: integer := 16; -- 2**w for LMS
    constant TREE_HEIGHT: integer := 10;

    constant BDS_K: integer := 8;

    constant HASH_CORES: integer := 5;
    constant HASH_CHAINS: integer := 4;
end package;
```

Listing 1. The configuration file allows to set various parameters before synthesis. Among others, it is possible to select the supported scheme, the BDS parameter, and the number of hash cores and OTS chains.

4.2 Agile Acceleration of XMSS and LMS

The similarities between XMSS and LMS make it possible to share many components between the accelerators of the two schemes, which reduces the overall hardware cost for an agile implementation. In the following, we describe the most important implementation aspects of the two schemes and our agile design.

For most modules, the signals for LMS and XMSS are the same. Since the access to the hash module is limited to the `WOTS` module and the `Thash` module, we can reuse the control logic for the remainder of the scheme (i.e., generating the tree structure and controlling the hash chain). The `WOTS` module and the `Thash` module determine the inputs to the hash function that differentiate between LMS and XMSS. Due to the reduced complexity of LMS, some signals are unused for LMS. The `Thash` module implements the L-Tree compressions and Merkle Tree hash operations for XMSS and only the Merkle Tree hash operations for LMS. It is controlled by either the `OTS` module (L-Trees), the `Path-Generator` module (Treehash), or the `Verify` module. Because of the different addressing schemes of the Merkle Tree nodes for LMS and XMSS, the most notable difference in inputs to `Thash` is that for LMS the node index is calculated using $(2^h + leaf\_index\_leftmost\_child)/2^{h_{node}}$ and for XMSS using $leaf\_index\_leftmost\_child/2^{h_{node}}$. The design uses two dual-port BRAMs, one as internal storage for the BDS *retain* array and as intermediate storage for the WOTS secret and public key values. The other BRAM is used as an I/O interface, with one port being accessible from outside the design. This BRAM is used to make the generated public key available after key generation and the WOTS signature and authentication path after signing. For verification, the public key and the signature must be written to the same memory locations. The design is split into two main controller modules, the `Path-Generator`, which implements the BDS algorithm (key and signature generation), and the `Verify` module, which implements only signature verification. The `Path-Generator` can divert control to the `OTS` module.

4.2.1 OTS Acceleration. As described in Section 3, the OTS schemes used in XMSS and LMS are both based on the WOTS scheme. However, XMSS uses the keyed hash function with masked inputs, while LMS only prepends a tweak value to the hash input. The *WOTS* module implements all operations of the OTS schemes. It features a seed expander following the seed expansion algorithms described in [34] and the XMSS reference implementation. The seed expansion for XMSS

does not follow the RFC, as it suffers from a multi-target attack [20].[2] The generated WOTS secret key is stored in the internal BRAM. In the `Chain-Compute` module, the WOTS chains are calculated using the internal BRAM as a buffer for the final chain values for key generation and verification, reading the start values from either internal BRAM or the IO BRAM. For signature generation, the signature values are written to the IO BRAM.

Arguably the biggest difference between XMSS and LMS is the way an OTS public key is compressed to a leaf node of the Merkle Tree. In LMS, all ℓ n-byte words are fed into the hash function sequentially, resulting in a single digest value. This process only requires a single hash invocation, albeit with a large message input. XMSS, on the other hand, uses L-Trees to compress the OTS public keys. The construction uses the same key and mask technique as in the treehash algorithm of XMSS. Because of these differences, the compression for the schemes is implemented in two different compressor modules. For LMS, the module first generates the LM-OTS public key and afterward uses the result to generate the leaf node, which uses its own distinct hash operation. The `L-Tree` module follows the algorithm described in [25], using the `Thash` module for the keyed hash function calls. No optimizations have been applied to *L-Tree*, since our analysis revealed that the L-Tree construction has only a small impact on the overall running time of the WOTS key generation algorithm.

4.2.2 Treehash Acceleration. The treehash implementation in the `Path-Generator` is divided into the `Initializer` and the `Treehash` module. A single stack is also instantiated, which is used by the two modules. The `Initializer` implements a generalization of the treehash routine described in the XMSS RFC, using the `WOTS` module to generate leaf nodes, as well as the stack and the `Thash` module to generate higher-level tree nodes. By exposing the current node, its location within the tree, and whether it is part of the authentication path or needs to be copied into a treehash instance or the *retain* array, the module allows the `Path-Generator` to copy the node into the respective register or—in the case of retained values—the internal BRAM. The retain array is specifically placed in BRAM to enable the configuration of larger k, for which the *retain* array would exceed the available FFs. The sequential treehash algorithm is implemented in `Treehash`, which stores the nodes for all treehash instances and can calculate a single treehash update operation at a time. While a parallel design would also be feasible, the design would need as many `WOTS` modules as it has treehash instances to effectively parallelize the leaf node generation. Moreover, each parallel treehash module would require its own stack, while the sequential design uses a single shared stack. For each instance, the current lowest height of the instance on the stack is stored in a register, such that the `Path-Generator` can select the correct treehash instance to start based on their heights. For key generation the `Path-Generator` generates the root node using the `Initializer` module. When signing, it uses the `WOTS` module to generate the WOTS signature for the given message hash, writes the current path to the IO BRAM, and then generates the upcoming path according to the BDS algorithm. Finally, the module searches for the treehash instance with the lowest height on the stack and then starts the `Treehash` module to work on this instance.

4.3 Hash-bus Implementation

The hash controller is responsible for making k hash cores available to the signature core over a predefined bus interface and is a key design element in our configurable implementation. The controller allows instantiating an arbitrary number of hash accelerators via a simple configuration file. We provide an implementation of SHA2-256; however, the abstraction of the hash implementation from the signature implementation through the hash controller logic allows straightforward

[2] https://github.com/XMSS/xmss-reference/commit/3e28db2362f25600699972766e7782635b1826f5.

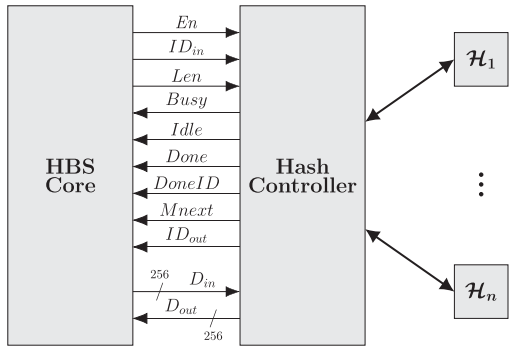


Fig. 5. Structural overview of the hash controller that manages the assignment of hash cores to requests from the bus.

design of extensions to support additional hash algorithms or use implementations with different optimization targets. For example, one could replace our SHA2 core with a SHAKE-128 implementation or switch out our performance-focused general-purpose implementation of SHA2 with an XMSS-specific optimized variant as proposed in [46].

Figure 5 depicts a structural overview of the interconnection of the hash accelerators and the signature core. The hash controller processes inputs in 256-bit blocks and requires unique *IDs* as input to map each input block to the corresponding hash operation. The input size of 256 bits allows it to feed an entire block in a single clock cycle. While XMSS only uses inputs that are multiples of 256 bits throughout the scheme, LMS does not have a fixed block size. The *len* signal indicates the length of the entire input. The interface of the hash controller features a *busy* and an *idle* signal. The busy signal indicates whether the hash controller is able to receive new hashing requests. If the busy signal is set, either all available hash cores are currently in use or the bus is occupied by an *mnext* request. Such *mnext* requests are sent when a hash core requires the next message block to continue the hash operation. To avoid unnecessary stall cycles, in such a case, the bus is automatically blocked to all actions other than serving the next message block by setting the *busy* signal to *high*. Simultaneously, the *ID* is forwarded on the ID_{out} port to identify the module the task originates from.

When the *busy* signal is set to low, new hash operations can be started from the XMSS core by setting the enable signal to high for the duration of one clock cycle, alongside an identifier, the length of the data input, and the first 256-bit block of the message. The hash controller then forwards the data to one of the idling hash cores. The padding that is required for the hash algorithms is implemented within the hash core. Our SHA-2 implementation allows arbitrary input lengths. Hence, it would also be feasible to make the hash-bus interface available to other resources on the chip. Th e signature core is not required to implement a hash-specific padding that facilitates the logical separation between the hash-core implementation and the signature core. As soon as a hash operation is finished, the output of the respective hash core is forwarded via the D_{out} port alongside the *ID*, which is provided via the *DoneID* signal. While the *ID* could have also been forwarded via the ID_{out} signal, this causes stall cycles if another hash core tries to set the *mnext* signal in the same cycle the respective hash core sets the *done* signal. By separating the ID_{out} signal from the *DoneID* signal, such conflicts can be avoided. However, it can still occur that two hash cores try to fetch the next message block in the same clock cycle. Such an *mnext-collision* is resolved by halting one of the hash cores for one clock cycle. These collisions happen very rarely and therefore hardly affect the performance of the scheme. Similarly, such collisions can arise when two hash

cores finish at the same time. Analogously to the handling of menxt-collisions, one of the hash cores is halted and the output is forwarded with one clock cycle delay.

It must be noted that the input side of the hash controller does not resolve bus communication conflicts automatically. In most cases, only one module on the signature-core side is active at a time, which makes conflict handling unnecessary; e.g., during the treehash algorithm, the internal state logic ensures that in each clock cycle no more than one hash operation is started. A special case arises in the parallel computation of WOTS+ chains, which is further examined in the following.

4.4 OTS Chain Parallelism

It is in no doubt that the performance of the deployed hash core(s) is a key factor in the overall performance of XMSS and LMS. Both schemes use the hash function extensively throughout the computation of the one-time signatures and the treehash algorithm. In addition to that, XMSS generates all bitmasks and hash-keys that are used throughout the scheme pesudorandomly using the hash function, which results in an even higher-capacity utilization of the hash accelerator. However, our analysis quickly revealed that simply adding a large number of hash cores is not sufficient to parallelize the schemes effectively since most of the hash operations depend on the result of prior operations. For example, during each step of the WOTS+ chaining algorithm of XMSS, one hash operation is required to generate the pseudorandom key and bitmask, respectively. These two operations are followed by the chain step, which uses the key and the masked value of the previous chain step as input. In LMS, this problem is even more severe since the hash operations of the LM-OTS chaining algorithm directly depend on the outcome of the previous hash operation. Hence, the opportunities for parallel hash computations are very limited. Other parts of the algorithm offer slightly more opportunities for naive parallelism.

In our hardware implementation, we take advantage of all the opportunities where parallel computation is feasible, e.g., by computing the bitmask and the hash key simultaneously. However since the OTS chaining algorithm is a crucial part in the performance of the HBS, especially during key and signature generation, it requires further acceleration. To allow optimal acceleration of the OTS chaining algorithms, we instantiate a configurable amount of OTS chaining modules in parallel. Since each OTS signature consists of many independent chains (using XMSS with a 256-bit hash function and $w = 16$, a signature is made from 67 WOTS+ chains), parallel computation of chains allows for much improved scheduling of hash cores with less idling due to dependencies. Due to the parallel chaining module, multiple modules could now write simultaneously on the hash bus, requesting the start of a new hash operation. Since the hash bus itself does not implement conflict resolution measures on the input side of the hash controller, a mechanism to avoid such conflicts is required. To keep the overhead low, we implement a **time-division multiplexing (TDM)** solution where each OTS chaining module is assigned to a fixed time slot during which new hash operations can be started. The TDM is implemented using a straight ring counter. A part of the *ID* that is used for the hash operation identifies the originating OTS chaining module. Hence, when the hash controller signals *mnext* to request the next message block, this request can be directly forwarded to the corresponding chaining module. The ring counter is then reset and continues in the next clock cycle with the subsequent chaining module.

5 EVALUATION

In this section we evaluate our implementation and compare it to related work. We start by exploring the performance limits of both schemes and investigating the effect of additional hash cores and OTS chaining modules. We then continue to analyze the area requirements for different configurations. During this analysis, we also examine the overhead of the agile implementation over a standalone XMSS design. Combining the results of the area and performance analysis, we

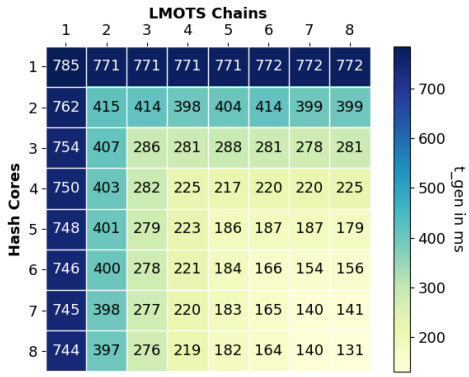


Fig. 6. Latency in *ms* for LMS key generation using $h = 10$, $w = 16$, and SHA2-256 with different number of configured hash cores and LM-OTS chaining modules.

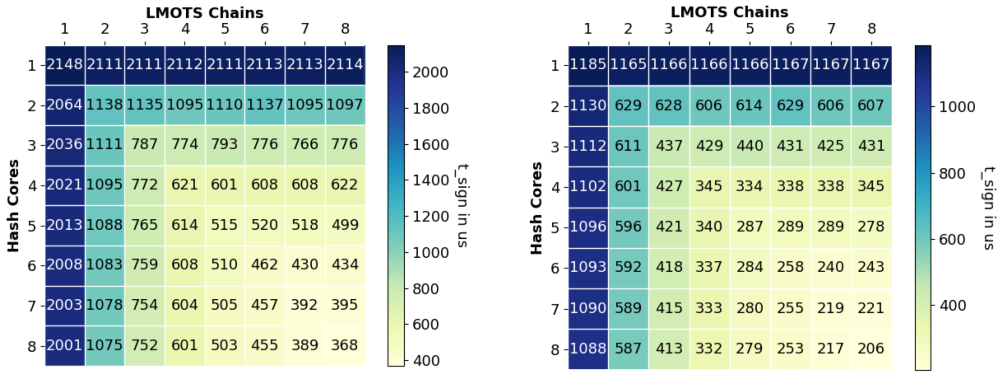
establish optimal time-area tradeoffs for the implementations with up to eight hash cores and eight OTS chaining modules. Finally, we discuss the results and compare the differences of XMSS and LMS.

Our design allows configuring an arbitrary number of hash cores and OTS chains, which allows various tradeoffs regarding area and performance. To analyze the impact of this configurability, we evaluate each aspect of the implementation using up to eight hash cores and eight parallel OTS chains. Unless otherwise stated, we use $h = 10$, $w = 16$, and $n = 32$ in the following. For the BDS parameter k, we explore $k = 2$ and $k = 8$. Using the former, very few nodes are stored in BRAM to accelerate the signature generation. The latter stores many Merkle Tree nodes in BRAM, which should lead to a significant increase in the performance of the signature generation. Choosing k close to the minimum and maximum yields a good intention for the design space and possible tradeoffs.

5.1 Performance Exploration

We start the evaluation by exploring the performance results of our implementation. Therefore, we examine LMS and XMSS separately. The timing properties are mostly independent of the configuration that places either one of the schemes or an agile implementation of both on the **Field Programmable Gate Array (FPGA)**. For now, we assume that every configuration can be placed with a frequency of 100 MHz. This allows us to estimate the execution time for all configurations in the following without having to execute the complex optimization strategies to establish the maximum clock frequency. We report the maximum frequency for some interesting configurations separately in Section 5.4.

5.1.1 LMS. The results for the performance of the LMS key generation are shown in Figure 6. Each cell of the figure holds the latency of the key generation for the respective number of hash cores and parallel LM-OTS chains. Using the minimal configuration with one hash core and one LM-OTS chaining module, the key generation takes 785 ms. From the figure, it is clear that only adding hash cores *or* only adding LM-OTS chaining modules barely improves the performance, that is, since each step in the hash chain operates on the output of the previous iteration. Hence, it is not possible to compute multiple steps in the same LM-OTS chain in parallel. Similarly, it is not feasible to compute multiple hash chains in parallel if only one hash core is available, that is, since the capacity utilization of the hash function is extremely high throughout the scheme; i.e., the timing overhead for the control logic is negligible compared to the hash computation.



(a) $k = 2$ (b) $k = 8$

Fig. 7. Average latency in *us* for LMS signature generation using $h = 10$, $w = 16$, and SHA2-256 with different number of configured hash cores and LM-OTS chaining modules. The left figure shows the latency with BDS parameter $k = 2$ and the right shows the latency with BDS parameter $k = 8$.

The figure shows that a 1:1 relation of hash cores and LM-OTS chains (the diagonal) yields the best performance results, and deviating from the diagonal in either direction barely improves the performance.

Figure 7 shows the latency of the signature generation for various configurations of hash cores and LM-OTS chains in microseconds. Since the latency for the signature generation varies slightly depending on the (hashed) input and the state of the BDS algorithm, the depicted results are averaged over the first 50 signatures. We verified experimentally for a small subset of configurations that the average over all signatures falls in the same range, and hence, the measured timing is a good approximation. The left figure depicts the results with BDS parameter $k = 2$ and the right with BDS parameter $k = 8$. The BDS parameter allows for a tradeoff between used BRAM memory and area and performance. Depending on the chosen parameter, the signature generation takes between 2,148 us and 1,185 us for the minimal configuration with one hash core and one LM-OTS chain. Hence, maintaining more intermediate nodes in BRAM can almost double the performance. Similar to the key generation, the 1:1 ratio of hash cores and LM-OTS chains yields the best performance results. The largest tested configuration generates signatures in 368 us and 206 us, respectively. It is possible to configure even more hash cores and LM-OTS chains, further reducing the latency. Overall, the signature generation outperforms the key generation by orders of magnitude, which is a testament to the effectiveness of the BDS algorithm. By storing intermediate nodes of the tree, the computation of the authentication path does not need to recreate the entire tree, which results in greatly improved performance. The initial storage of intermediate nodes occurs during the key generation phase, which automatically stores the first authentication path. With each signature, the BDS algorithm precomputes and stores nodes that will be required for later signatures.

The latency for the verification algorithm is shown in Figure 8. Using a single hash core and LM-OTS chain, the verification algorithm takes 371 us. Again, the configurations with equal number of hash cores and LM-OTS chains yield the best results. This is expected since the verification algorithm on average needs to compute each hash chain for half the overall length and then needs to combine the computed OTS public key with each node of the authentication path separately. The largest configuration tested in this evaluation verifies a signature in about 85 us.

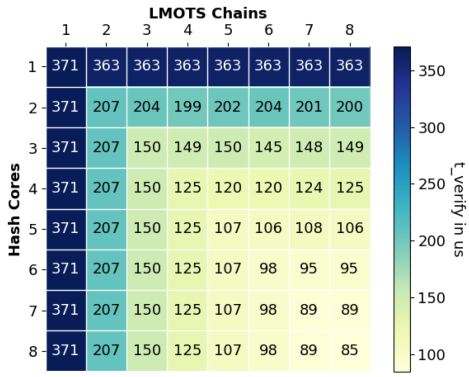


Fig. 8. Average latency in *us* for LMS signature verification using $h = 10$, $w = 16$, and SHA2-256 with different number of configured hash cores and LM-OTS chaining modules.

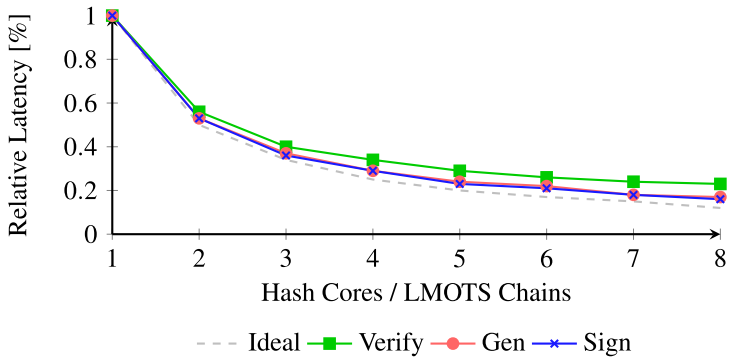


Fig. 9. Relative latency of LMS as a function of the configured hash cores and LM-OTS chains in comparison to the configuration with only one hash core and LM-OTS chain. Lower is better.

A different aspect that is visible in all of the prior performance figures is that the largest performance gaps occur for a relatively low number of hash cores and LM-OTS chains. For key and signature generation as well as for the verification algorithm, the addition of a second hash core and LM-OTS chain almost doubles the performance. The effect reduces for a larger number of cores and parallel chains. This is natural since adding a second hash core and LM-OTS chain would ideally lead to a halved runtime. However, adding a third hash core and LM-OTS chain can only cut the running time of the minimal configuration into thirds, which does not halve the running time with two cores and OTS chains. Figure 9 shows the speedup for configurations with x hash cores and LM-OTS chains over the minimal configuration of LMS. The dashed line shows the best-case performance when each added hash core/LM-OTS chain would be utilized ideally; i.e., a second pair of chains/cores would double the performance. Both the key and signature generation function come very close to the ideal speedup. The verification algorithm achieves slightly worse performance compared to the ideal boundary. The reason for this can be found in the algorithm itself: In the first phase, the LM-OTS public key is computed. This can be parallelized similarly to the key and signature generation. In the second phase, the computed LM-OTS public key is iteratively combined with the authentication path nodes to compute the root node of the Merkle Tree. In this phase, the additional hash cores and LM-OTS chains cannot be utilized. For signature verification,

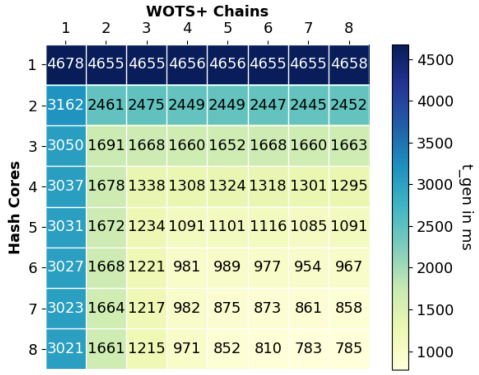


Fig. 10. Latency in *ms* for XMSS key generation using $h = 10$, $w = 16$, and SHA2-256 with different number of configured hash cores and WOTS+ chaining modules.

the percentage of time spent in the first phase is much lower than for key and signature generation. Thus, the acceleration for signature verification is slightly less effective.

5.1.2 XMSS. We now instantiate the XMSS algorithm and evaluate the performance. The results for the key generation are shown in Figure 10. The key generation for the configuration with one hash core and one WOTS+ chaining module takes about 4.7 s. By only adding a second hash core, this time can be reduced by about one-third. However, configuring even more hash cores does not improve performance significantly, that is, since in XMSS each WOTS+ chain iteration computes a pseudorandom key and mask using the hash function. Then, the previous hash chain value is masked and hashed using the key. The generation of the key and the mask value can be parallelized using two hash cores. However, the actual hash chain step can only be computed when the key and the masking value are known. Hence, further available hash cores cannot be utilized. From the figure, it is also clear that the effect of multiple WOTS+ chaining modules only starts showing with two or more hash cores. This is similar to the observations made for LMS. As opposed to the observation made for LMS, for XMSS the ideal relation between hash cores and WOTS+ chains is not 1:1, that is, since due to the extensive use of masks and pseudorandom keys, the design is more dependent on the hash function than on the number of WOTS+ chains. Due to the dynamic scheduling of hash cores to OTS chains and the different opportunities for parallelism during OTS chaining and treehash, there is no sharp ideal ratio as can be seen in the figures.

The average results for XMSS signature generation are shown in Figure 11 for BDS parameter $k = 2$ (left) and $k = 8$ (right). The characteristics regarding the relation between hash cores and WOTS+ chains are similar to the key generation. For XMSS, the largest configuration with eight hash cores and eight WOTS+ chains does not yield a desirable configuration as it only performs as well or slightly better then the configuration with eight hash cores and five WOTS+ chaining modules. Considering the area cost of additional hash cores, choosing the largest configuration is not a sensible choice. We further explore tradeoffs between the area cost and the achieved performance in Section 5.3. In the minimal configuration, the signature generation takes on average 12.6 ms for $k = 2$ and 6.9 ms for $k = 8$. This is a significant speedup over the key generation, again demonstrating the effectiveness of maintaining intermediate nodes in BRAM. As for LMS, the performance using $k = 8$ is almost doubled. Configurations with further hash cores and WOTS+ chaining modules increase the performance over the minimal configuration. Using eight hash cores and five WOTS+ modules, the signature generation takes 2.3 ms and 1.3 ms, respectively.

(a) $k = 2$ (b) $k = 8$

Fig. 11. Average latency in *ms* for XMSS signature generation using $h = 10$, $w = 16$, and SHA2-256 with different number of configured hash cores and WOTS+ chaining modules. The left figure shows the latency with BDS parameter $k = 2$ and the right shows the latency with BDS parameter $k = 8$.

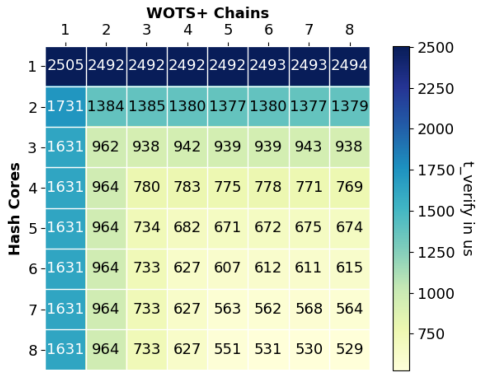


Fig. 12. Average latency in *us* for XMSS signature verification using $h = 10$, $w = 16$, and SHA2-256 with different number of configured hash cores and WOTS+ chaining modules.

The performance results for XMSS signature verification are shown in Figure 12. Similar to LMS, the XMSS verification is the fastest part of XMSS. Using the minimal configuration with one hash core and one WOTS+ chain, the verification algorithm has an expected latency of about 2.5 ms. In the boundaries of our evaluation, the fastest signature verification was achieved at 529 us using eight hash cores and eight WOTS+ chains. However, as discussed above, this configuration is not ideal since the much smaller configuration with five parallel WOTS+ chains and eight hash cores achieves similar performance with 551 us.

The performance of XMSS as a function of the configured Hash Cores and WOTS+ chains is plotted in Figure 13. Since XMSS does not yield ideal configurations with a 1:1 relation of hash cores and WOTS+ chains, we use the fastest measured time at a fixed number of hash cores for the evaluation. The measured latency is close to the ideal speedup that can be achieved by adding more hash cores. Overall, speedups of up to 80% can be achieved. As for LMS, the verification algorithm performs slightly worse compared to key and signature generation. However, the additional hash cores can be utilized slightly better in XMSS, which is a result of the additional hash keys and masks that are computed during the WOTS+ verification as well as the treehash operation.

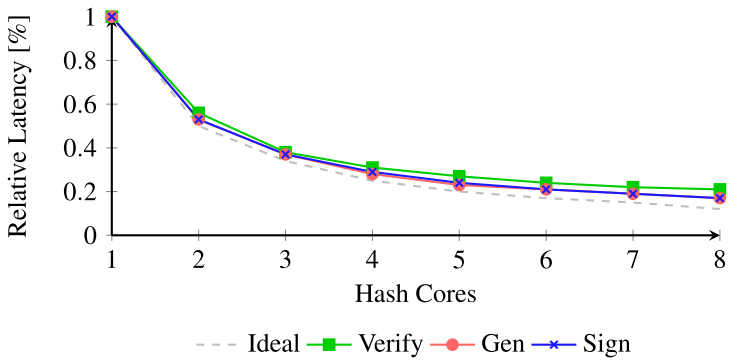


Fig. 13. Relative latency of XMSS as a function of the configured hash cores in comparison to the configuration with only one hash core. Lower is better.

Table 2. Performance Comparison of Our Hardware Implementation of XMSS and LMS Using the Minimal Configurations

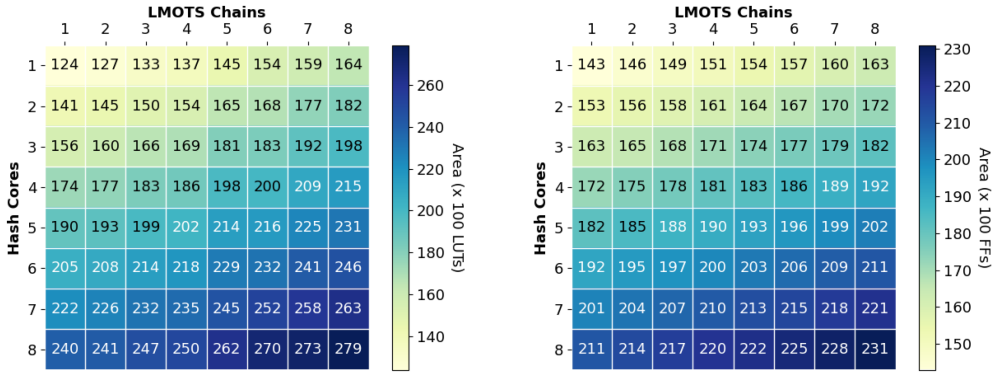
	Keygen	**Sign**	**Verify**
LMS	785 ms	2.1 ms	371 us
XMSS	4.68 s	12.6 ms	2.5 ms
Difference	**x 6.19**	**x 6**	**x 6.74**

5.1.3 Comparison. To finalize the performance analysis, we compare the results of the two schemes. As a result of the pseudorandom bitmasks and hash keys, XMSS requires far more hash operations, which leads to a much longer running time throughout all XMSS functions compared to LMS. In Table 2 we list the latencies for keygen, signature generation, and verification for both schemes in the minimal configuration using only one hash core. Overall, we find that XMSS is about 6 times slower than LMS for each operation. This is due to the increased complexity of XMSS (see Figures 1 and 3). Another factor that contributes to the performance difference is the block length of the hash function. Since XMSS uses larger inputs to the hash function than LMS, the hash computations in XMSS take longer than in LMS. This effect can be reduced by using a hash function with larger block length like SHAKE.

5.2 Area Utilization

We now evaluate the hardware requirements for both schemes. We start by separately analyzing the standalone versions of LMS and XMSS. Then, we instantiate the agile version that supports both schemes and analyze the overhead. For the synthesis, we use Xilinx Vivado 2021.2 with an Artix-7 target device (specifically the `xc7a75tcsg324-2L`). The FPGA features 47,200 LUTs and 94,400 FFs in total. In the following, we report the area utilization after synthesis using the area-optimized synthesis strategy of Vivado for evaluation-time reasons. We report the numbers after place and route as well as the achieved frequency for some configurations in Section 5.4.

The BDS parameter k selects how many nodes of the Merkle Tree are stored in BRAM during the key generation. For smaller k, only a few nodes are stored, while for large k, many nodes are stored. We showed the effect on the performance in the previous section. However, quantifying the area cost is more complicated since the parameter mostly affects the amount of BRAM used and barely changes the area utilization measured in Lookup Tables (LUTs) and FFs. Our implementation instantiates a BRAM that is sufficiently large for $k = 2$ and $k = 8$ independent of the actual



(a) LUTs (b) FFs

Fig. 14. Area utilization measured in LUTs and FFs of the implementation that only supports LMS. The configuration uses $h = 10$, $w = 16$, $k = 2$, and SHA2-256 as hash function.

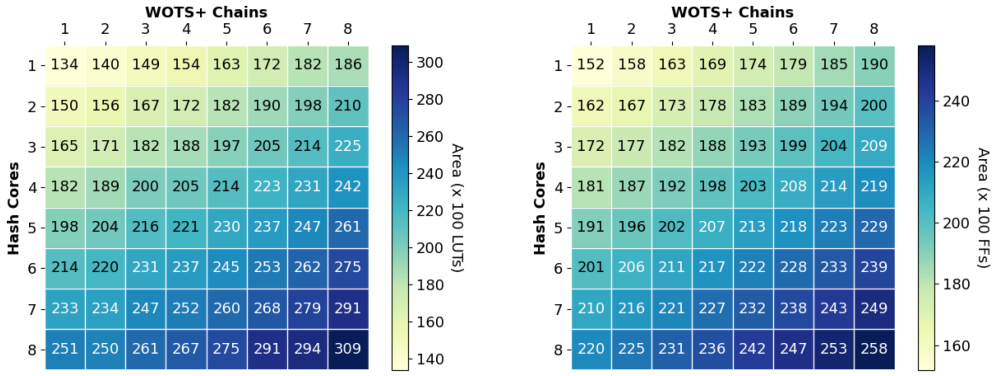
value for k. For $k = 8$, we need about 10 kB of BRAM, while for $k = 2$ only 2.2 kB BRAM storage is used. The tradeoff between area, performance, and BRAM usage needs to be considered for a given use-case and is especially interesting for large tree heights where it may not be possible to store all nodes in BRAM. In the following analysis, we focus on $k \in \{2, 8\}$ and report the LUT and FF utilization. Other values for k can be chosen in the configuration file.

5.2.1 LMS. We start by analyzing the area requirements for LMS. The results for the area utilization measured in LUTs as a function of the configured hash cores and LM-OTS chaining modules is shown in Figure 14(a). The minimal configuration uses about 12,000 LUTs, while the area requirements for the largest configuration is more than double at about 28,000 LUTs. On average, adding an additional hash core adds 1,475 LUTs to the design, while adding an additional LM-OTS chaining module adds about 519 LUTs. Hence, in terms of hardware utilization, additional LM-OTS chains are much cheaper. The BDS parameter k has only a small influence on the LUT utilization as can be seen in the difference between the left and the right figure. The LUT utilization for $k = 8$ is similar to the configuration with $k = 2$.

Figure 14(b) shows the FF utilization for LMS with $k = 2$. The smallest design can be placed with 1,430 FFs. Each additional hash core requires on average 1,156 FFs and each additional LM-OTS chaining module requires on average 563 FFs. This is similar to the results using LUTs as a metric. The configuration with $k = 8$ uses fewer FFs than the configuration with $k = 2$, that is, since for smaller k more FFs are needed as internal storage for the additional treehash instances. On average, the configuration with $k = 2$ uses about 10% more FFs than the configuration with $k = 8$.

5.2.2 XMSS. We now analyze the area requirements for the XMSS-only implementation. The results for the LUT utilization are shown in Figure 15(a). Compared to LMS, the XMSS implementation uses slightly more LUTs. Like in the LMS analysis, the LUT utilization remains similar when using $k = 8$ instead of $k = 2$. Each additional hash core increases the LUT utilization on average by 1,500 LUTs. Adding a WOTS+ chaining module on average results in 713 added LUTs. The size estimate for the hash core is consistent with the LMS evaluation, which resulted in almost equal size. For the OTS scheme, it shows that WOTS+ requires more LUTs than LM-OTS, which is expected due to the generation of the additional key and bitmask.

The FF-based area evaluation for XMSS is shown in Figure 15(b). As was the case in LMS, the configuration with $k = 2$ uses more FFs compared to $k = 8$. Each added hash core increases the



(a) LUTs (b) FFs

Fig. 15. Area utilization measured in LUTs of the implementation that only supports XMSS. The configuration uses $h = 10$, $w = 16$, $k = 2$, and SHA2-256 as hash function.

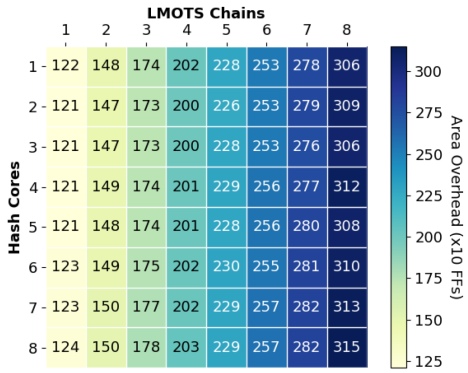


Fig. 16. Area overhead of the agile implementation over the LMS-only version measured in FFs.

area usage by 1,175 FFs and each WOTS+ chain by 681 FFs. The results for the WOTS+ chains indicate increased complexity compared to LM-OTS, which required 563 FFs per chain module.

In general, the hardware utilization for XMSS is slightly increased over LMS. There are a few reasons for this: First, we already identified that the WOTS+ module is more complex than the LM-OTS module. Especially for multiple parallel chaining modules, this (small) overhead adds up and contributes to the overall size. Second, the compressing mechanism for the OTS public keys in LMS is much simpler than the L-Tree approach of XMSS. In the configuration with only one WOTS+ chain, the L-Tree compression mechanism contributes most to the increased area utilization.

5.2.3 Agile Version. In the previous sections we analyzed the standalone versions of LMS and XMSS. Usually when a device is required to support both XMSS and LMS, one would need to instantiate two hardware accelerators, each with the area utilization described above. However, due to the similarities of LMS and XMSS, much of the hardware logic can be reused to minimize the overhead. In this section we examine the hardware overhead that occurs when configuring the agile version.

We first measure the area overhead of the agile version over the standalone LMS implementation. Figure 16 shows the absolute overhead of the agile implementation measured in FFs. The same

Table 3. Comparison of the Hardware Utilization for LMS, XMSS, and the Agile Instantiation after Place and Route on a Xilinx Artix 7 FPGA

	Logic		Memory			Usage
	LUT	MUX	FF	LUTRAM	BRAM Util.	Slices
Only LMS ($k = 2$)	12,407	373	13,293	32	2.2 kB	4,683
Only XMSS ($k = 2$)	12,986	309	14,203	32	2.2 kB	4,661
Agile LMS & XMSS ($k = 2$)	15,405 (+19%)	537 (+44%)	14,512 (+2%)	32 (±0%)	2.2 kB	5,115 (+10%)
Only LMS ($k = 8$)	12,374	0	11,630	32	10.1 kB	4,450
Only XMSS ($k = 8$)	12,038	421	12,535	32	10.1 kB	4,887
Agile LMS & XMSS ($k = 8$)	14,615 (+21%)	515 (+22%)	12,854 (+3%)	32 (±0%)	10.1 kB	4,528 (−9%)

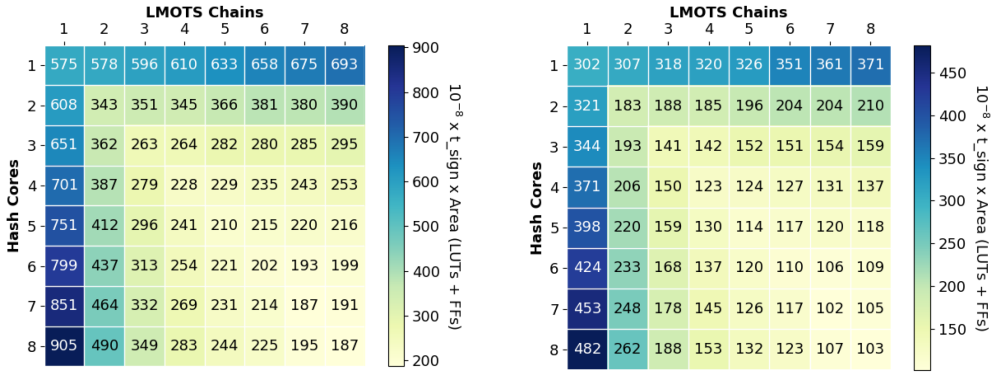
All designs use $h = 10$, $w = 16$ with SHA2-256 and are placed with 100 MHz using Vivado's area-optimized strategies and use a single hash core with a single OTS chaining module. The relative overhead is computed in comparison to the XMSS-only version.

evaluation using LUTs as a metric yields similar results. The figure confirms the observation made earlier, that the WOTS+ chaining modules are slightly larger than the LM-OTS chaining modules. Since the OTS chaining modules in the agile implementation supports both LM-OTS and WOTS+, the agile OTS module is also larger than the LM-OTS-only module, which leads to the increasing overhead on the chain-module axis. We also measure the overhead of the agile implementation over the XMSS-only design and find that the overhead is almost constant for all numbers of hash cores and OTS chaining modules. Hence, there is no gradual increase in overhead of the agile implementation over XMSS for larger configurations as there is with LMS. That is, since most of the LMS scheme can be realized by skipping states of XMSS, e.g., by omitting the key and mask steps in the WOTS+ scheme, LM-OTS can be realized. Hence, the overhead to XMSS comes mostly down to additional edges in the state machines. However, few aspects of LMS, like the LM-OTS public key compression, cannot be realized in this way, which leads to a small increase in size.

Table 3 shows the utilization report of the minimal configurations of LMS, XMSS, and the agile version after place and route on the Artix 7 FPGA. It further shows the relative overhead of the agile design compared to the standalone versions of LMS and XMSS. For $k = 2$ and $k = 8$, the agile design has a 20% increased LUT utilization. Moreover, the design uses many more multiplexers than the standalone versions of LMS and XMSS. The reason for this increase is that at many occasions conditional statements are implemented that switch between the two schemes. For the standalone version, these statements are removed by optimizations during the synthesis process. The register overhead of the agile design over XMSS is low at about 3%. The LMS version uses slightly fewer FFs, which is due to the simpler algorithm. The agile design essentially uses the same registers as LMS and XMSS. Due to the similarities, almost all registers can be shared between the schemes, which leads to the minor increase in FFs. For the slice utilization, the results are inconclusive with a 10% increase for the agile design using $k = 2$ and an almost 10% decrease for $k = 8$. This is an artifact of the largely undocumented place and route algorithm used by Vivado.

5.3 Time-area Optimization

So far, we have analyzed the performance capabilities and the area utilization of our implementation separately. For some use cases, this is the best way of choosing a configuration, e.g., if there are hard time constraints like "*the key generation must be completed in less than X seconds*" or if the available space is limited and the performance should be maximized. In the former case, one would choose the smallest configuration that meets the time constraints, and for the latter, the largest configuration that fits the available area. However, in reality, there often is a tradeoff between area and performance to be made. Since space on the hardware design is expensive in the



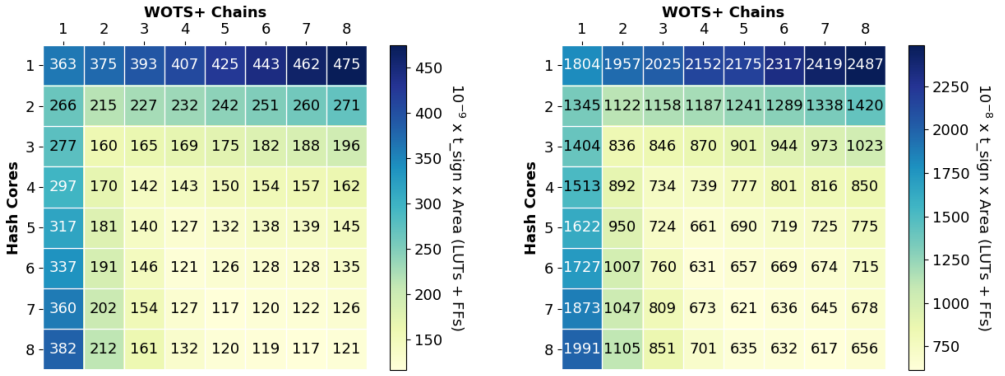
(a) $k = 2$ (b) $k = 8$

Fig. 17. Time-area product of the implementation that only supports LMS. Lower is better. The configuration uses $h = 10$, $w = 16$, and SHA2-256 as hash function. The left figure shows the results for BDS parameter $k = 2$ and the right figure shows $k = 8$.

manufacturing process, engineers often like to use the smallest design that yields the best performance. In this section, we aim to find such optimal tradeoffs for the standalone designs of LMS and XMSS as well as the agile design. To do this, we take the area results (LUTs + FFs) for each configuration and multiply them by the average latency of a signature generation. We omit the area influence of the BRAM since, while the amount of memory required changes with the BDS parameter k, it does not affect the number of hardware BRAMs utilized on the FPGA. The effect is only reflected in the FPGA BRAM utilization for larger tree heights or for ASIC targets. We use the sign algorithm for the optimization since often, this is the most timing-critical part of the signature scheme. To keep the numbers of the time-area product in a readable region, we further multiply a scale factor between 10^{-8} and 10^{-9}.

5.3.1 LMS. Figure 17 shows the result for the time-area product for configurations of LMS with up to eight hash cores and eight LM-OTS chaining modules. The results for BDS parameter $k = 2$ (left) and $k = 8$ (right) are very similar. In the performance analysis (Section 5.1), we already found that for LMS, a relation of one hash core per LM-OTS chaining module is best to accelerate the scheme. Combined with the almost linear area overhead per added hash core and LM-OTS chaining module, configurations that deviate from the diagonal line generally yield a higher time-area product. For $k = 8$ (right), the time-area product is minimal at seven hash cores and seven parallel LM-OTS chains. We expect this to be the global minimum for the time-area product since even larger configurations only slightly increase the performance (c.f. Figure 7), but the area increases linearly with each added hash core and LM-OTS chain (c.f. Figure 14(a)). For $k = 2$ (left), the time-area product of the configuration with seven hash cores and seven LM-OTS chains is almost equal to the configurations with eight hash cores and eight LM-OTS chains. However, without the scale factor, the 7x7 configuration yields a slightly lower time-area product. Hence, we conclude that for the LMS-only design, a configuration with seven hash cores and seven LM-OTS chaining modules minimizes the time-area product.

5.3.2 XMSS. We repeat the analysis for the design that only implements XMSS, the results of which are shown in Figure 18. As opposed to the LMS evaluation, the ideal relation between hash cores and parallel OTS chains is not 1:1, which leads to a more diffuse distribution of configurations with low time-area product in the figure. However, for both $k = 2$ and $k = 8$, the lowest time-area



(a) $k = 2$ (b) $k = 8$

Fig. 18. Time-area product of the implementation that only supports XMSS. Lower is better. The configuration uses $h = 10$, $w = 16$, and SHA2-256 as hash function. The left figure shows the results for BDS parameter $k = 2$ and the right figure shows $k = 8$.

product occurs with eight hash cores and seven WOTS+ chains. For $k = 2$, the configuration with seven hash cores and five WOTS+ chains achieves almost the same time-area product. Without the scale factor, the 8x7 configuration performs slightly better. There may be configurations with even lower time-area products outside of our 8x8 evaluation scope. However, we do not expect these to improve the product by much since the results are flattening out in the lower right edge of the figure. Similar to LMS, the impact of additional hash cores and WOTS+ chains reduces with each added core/chaining module, while the hardware overhead is almost constant for each added module.

5.3.3 Agile Version. The agile design combines the two schemes at the cost of a small hardware overhead. The timing results are mostly unaffected by these changes. Since the LMS-only and the XMSS-only designs yielded very similar configurations that minimize the time-area product, it is unsurprising that these configurations lead to a small time-area product in the agile design. We verified this intuition using the same technique as above.

5.4 Discussion and Comparison

We now summarize our key findings and compare how LMS compares to XMSS and the agile implementation. We select configurations of interest for which we determine the maximum clock frequency supported by the design and report the area utilization after place and route on the target device. Finally, we compare our results to related work and discuss the results.

Overall, our evaluation showed that parallelizing hash operations is an effective measure to accelerate both schemes. However, we also found that the parallel hash cores can only be utilized efficiently if the OTS chaining algorithm can also be parallelized. For LMS, the optimal relation of OTS chains to hash cores is 1:1. XMSS, on the other hand, requires more hash cores than OTS chains for ideal acceleration. That is due to the extensive generation of hash keys and bitmasks, which can be done in parallel. Regarding the area requirements, we found that the standalone versions of XMSS and LMS are fairly similar. However, in all investigated configurations, XMSS had slightly higher area utilization, which is a result of the used keyed hash functions and the more complicated compression of the leaf nodes. Moreover, the WOTS+ hash chains are slightly larger than for LM-OTS. This leads to an incremental overhead of XMSS over LMS for configurations

Table 4. Comparison of Our Results in Different Configurations Compared to Themselves and Related Work

Scheme/ Variant (Cores, Chains)	Logic	Memory		FMax	Gen	Sign	Verify
	LUT	FF	BRAM				
XMSS [36, 46] HW-SW Co-design	6,289	8,579	16	93 MHz	3.44 s	9.95 ms	5.68 ms
LMS Gen [42]	190,660	126,656	17	285 MHz	3.06 ms	-	-
XMSS Gen / Sign / Vrfy [12]	8,464	14,464	0	110 MHz	2.88 s	11.09 ms	1.45 ms
	22,300	31,215	0				
	6,979	10,860	0				
XMSS (Ours, [44]) Minimal (1x1)	7,177	3,027	14.5	100 MHz	4.63 s	10.33 ms	2.68 ms
XMSS (Ours, [44]) Time-Area (3x2)	12,463	6,525	14.5	100 MHz	1.68 s	4.98 ms	1.01 ms
XMSS (Ours, [44]) Performance (9x7)	30,454	15,191	14.5	95 MHz	0.77 s	4.20 ms	561.01 us
XMSS / LMS / Both (Ours) BDS $k = 2$, Minimal (1x1)	13,630	15,278	15	100 MHz	4.68 s	12.6 ms	2.5 ms
	13,137	14,365	15	100 MHz	785 ms	2.1 ms	371 us
	15,946	15,566	15	100 MHz	— same as above —		
XMSS / LMS / Both (Ours) BDS $k = 2$, Medium (5x4)	22,833	20,789	15	100 MHz	1.1 s	3.0 ms	682 us
	22,033	19,125	15	100 MHz	223 ms	682 us	125 us
	27,032	21,228	15	98 MHz	— x 1.024 —		
XMSS / LMS / Both (Ours) BDS $k = 2$, Large (8x7)	30,057	25,371	15	100 MHz	783 ms	2.1 ms	530 us
	28,533	22,903	15	99 MHz	141 ms	392 us	90 us
	36,315	25,808	15	81 MHz	— XMSS: x 1.234 ; LMS: x 1.222 —		
XMSS / LMS / Both (Ours) BDS $k = 8$, Minimal (1x1)	13,308	13,625	15	100 MHz	4.68 s	6.9 ms	2.5 ms
	12,698	12,708	15	100 MHz	785 ms	1.18 ms	371 us
	15,860	13,914	15	100 MHz	— same as above —		
XMSS / LMS / Both (Ours) BDS $k = 8$, Medium (5x4)	22,560	19,157	15	100 MHz	1.1 s	1.6 ms	682 us
	21,388	17,433	15	100 MHz	223 ms	340 us	125 us
	26,828	19,515	15	100 MHz	— same as above —		
XMSS / LMS / Both (Ours) BDS $k = 8$, Large (8x7)	30,643	23,723	15	100 MHz	783 ms	1.2 ms	530 us
	21,388	17,433	15	100 MHz	140 ms	217 us	89 us
	36,534	24,197	15	81 MHz	— x 1.234 —		

All implementations use h = 10, n = 32, and w = 16. The configurations of the implementation described in this work have been synthesized and placed on an Artix-7 FPGA (`xc7a75tcsg324-2L`) using Vivado's performance-optimized algorithms. The "/"-character separates lines in a table row.

with parallel OTS chains. For the agile design that implements LMS and XMSS simultaneously, the hardware overhead comes down to roughly 20% increased LUT utilization compared to standalone XMSS. The amount of FFs used is only slightly increased.

Table 4 shows the results of our implementation in different configurations compared to related work. We ran the synthesis and place and route algorithms using the performance-optimized strategies of Vivado for a Xilinx Artix-7 target (`xc7a75tcsg324-2L`). Almost all configurations can be placed at a clock frequency of 100 MHz. However, for the largest configurations with the agile configuration, the placement strategies reach a limit, which causes a much-reduced clock frequency for these two configurations. Our design instantiates a fixed-size BRAM to support all values for

$k \leq 8$. Each BRAM module of the FPGA has a word width of 32 bits. To support the 256-bit data width of the hash function, multiple BRAM modules are concatenated. This leads to the static amount of BRAM reported for our implementations. However, the versions with $k = 2$ utilize much less BRAM, which is relevant if the design is placed on custom hardware.

The agile implementation described in this work builds on our previously reported standalone XMSS implementation [44]. Compared to the standalone version, the agile implementation utilizes more hardware resources. The reason for that is that the standalone version heavily relies on the BRAM as it outsources the treehash stack in BRAM. It also does not implement the BDS algorithm but instead caches all leaf nodes during key generation in BRAM. However, this approach scales badly for large tree heights. Furthermore, we found that the signature generation, which is arguably the most critical operation of the design, could not utilize the additional hash cores and OTS chains well using the naive treehash approach. Therefore, this work implements the BDS algorithm, which largely contributes to the increased area utilization but limits the BRAM usage for any tree height and allows accelerating the signature generation more easily. The performance results for the key generation are similar to the ones of the standalone XMSS version. However, the signature generation of XMSS is much faster, leveraging the benefits of the BDS algorithm. Using $k = 2$ (and hence storing only 2.2 kB), the signature generation is almost as fast as with the naive treehash that stores all 1,024 leaf nodes (12.6 ms vs. 10.33 ms). Using $k = 8$, more nodes are stored by the BDS algorithm and the average signature latency reduces to 6.9 milliseconds for the smallest configuration. The signature generation of the large configuration with eight hash cores and seven WOTS+ chains outperforms the naive treehash implementation by a factor of two for $k = 2$, despite storing much fewer nodes in BRAM. For $k = 8$, the signature generation can be further accelerated and outperforms the naive version by a factor of four. The fact that our new implementation is better suited to accelerate the signature generation than the old one also shows at the time-area optimization. For our previous implementation, the optimal tradeoff was at three hash cores and two WOTS+ chains. For the new implementation, the minimal time area product was at eight hash cores and seven WOTS+ chains, indicating that the performance improvements for such large configurations outweigh the area overhead.

Compared to the hardware/software co-design in [36, 46], the configurability of our implementation allows various tradeoffs regarding area and performance. While our minimal configuration is slightly slower than theirs, configuring only a second hash core makes up the entire difference. Larger configurations of our implementation allow further acceleration of the scheme, outperforming the hardware/software co-design approach substantially. Moreover, the full hardware design does not require additional computing resources on an external CPU, which may be occupied by other tasks. It must be noted that the transfer of the results (e.g., the signature) from the hardware module to the main memory is not considered in our timing measurements. The latency of this step depends on the target device where our accelerator is implemented (i.e., the bus width from the I/O BRAM to the memory management). Using DMA, this additional latency can be avoided entirely. However, this is not always available on low-end devices. The hardware implementation of XMSS presented in [12] is split into distinct modules for keygen, sign, and verify. Our design combines all three operations and the medium configuration, which is comparable in size, outperforms their version by more than a factor of two. The accelerator for the key generation of LMS presented in [42] targets a different use case. Their design is much larger than ours, but therefore accelerates the LMS key generation beyond the configuration boundaries of our design. While their design targets server-grade CPUs, our focus is on embedded devices. What distinguishes our implementation from all previously mentioned is the agility and vast configurability. This is the key feature of our design and allows tailored tradeoffs for a large number of applications and use cases.

Table 5. Size and Performance of the Verification Unit

Scheme	Logic	Memory		FMax	Latency
	LUT	FF	BRAM		
XMSS	4,885	3,379	15	100 MHz	2.5 ms
LMS	3,914	2,454	15	100 MHz	0.4 ms
Agile	6,334	3,669	15	100 MHz	– same as above –

Each configuration is equipped with a single hash core and a single OTS chaining module. Larger configurations are possible. The latency is averaged over multiple signature verifications.

5.5 Verification-only Accelerator

There are many applications where only signature verification is needed. Especially in an embedded environment, it can make sense to reduce the hardware accelerator to a variant that can only perform signature verification. This could, for example, be a medical device that can receive over-the-air updates. In this scenario, the hardware unit acts as a trust-anchor that is verified by the device manufacturer. The update file could be signed using XMSS or LMS and the hardware verification unit can verify that the update is legitimate. The reduced hardware module has two advantages: first, it requires less area, which reduces the production cost, and second, the cost for certification is less due to the reduced complexity.

We implemented such a reduced version of our accelerator. The configurability is not affected by this; i.e., it is still possible to instantiate XMSS, LMS, or the agile version. Moreover, it is still possible to configure multiple hash cores and OTS chains for the verification-only variant. The area and performance figures are shown in Table 5. As expected, the timing behavior is unaffected by the smaller design. However, the area utilization is reduced significantly. As for the full design, the agile implementation has almost no overhead in FFs over the XMSS-only version. The LUT utilization of the agile version is increased by 30% over XMSS. Hence, the overhead is slightly higher than for the full implementation. The verification-only variant of LMS is particularly small in hardware. Compared to the full accelerator, the LMS verification requires only 30% of the LUTs and 17% of the FFs.

5.6 Possible Attacks and Countermeasures

The security of the primitives XMSS and LMS is based on hash functions and therefore well understood and believed to be a conservative choice. However, many real-world attacks do not target the cryptographic primitive but the implementation instead. So-called side-channel attacks [31, 32] may exploit the power consumption, the electromagnetic radiation, or timing characteristics of the implementation. Fault attacks [8], on the other hand, actively tamper with the device, e.g., by hitting the circuitry with a laser or by undervolting the device during (parts of) the computation. Due to the random nature of the hash function, the computations in XMSS and LMS have very little dependency on the key that can be attacked using side channels. However, in [29, 47], power side-channel attacks on the SHA-2-based pseudorandom number generator used in XMSS are presented. Generally, fault attacks pose a high risk on stateful signature schemes. That is, if the attacker is able to fault the internal state, they can easily forge signatures. Hence, the state needs to be protected carefully if physical attackers are within the threat model. Compared to a software implementation or even a hardware/software co-design like [46], our hardware implementation maintains the state completely in hardware, which prevents any attacks from the software level like out-of-bound writes, which could tamper with the HBS state. For protection on the hardware level, one could maintain several copies of the state and check for consistency. Since it is difficult

for an attacker to cause the same faults on several chip locations, attempts to tamper the state could be detected. Apart from attacking the state, fault attacks presented in [1, 13] can potentially be applied to the multi-tree variants of XMSS and LMS.

6 CONCLUSION

In this work we presented the first agile hardware accelerator for the stateful hash-based signature schemes LMS and XMSS. Leveraging the similarities of both schemes, our implementation incurs only a small area overhead compared to the standalone versions of XMSS and LMS. With the easy configurability during synthesis, one can choose to instantiate either one of the schemes or the agile version that supports both. We implemented the BDS algorithm and made the parameter k configurable to allow for various tradeoffs between BRAM usage and the performance of the signature generation. Furthermore, it is possible to configure an arbitrary number of parallel hash cores and OTS chaining modules, which communicate over a bus interface.

We explored the design space of our implementation regarding speed and area. Moreover, we analyzed the time-area product and derived configurations that optimize this. Throughout our work, we explored differences between LMS and XMSS. We found that XMSS is much slower and slightly more complex in hardware. In return, XMSS offers a higher-security model. Having an agile implementation of both schemes allows engineers to support both standards and therefore maintain interoperability between devices from different manufactures. Moreover, it is possible to switch between schemes if one of them becomes insecure in the future.

REFERENCES

[1] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. 2020. FPGA-based SPHINCS$^+$ implementations: Mind the glitch. In *23rd Euromicro Conference on Digital System Design (DSD'20)*. IEEE, 229–237. https://doi.org/10.1109/DSD51259.2020.00046

[2] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[3] J.-P. Aumasson, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, and P. Schwabe. 2019. SPHINCS+-Submission to the 2nd round of the NIST post-quantum project. *NIST PQC Round* 2 (2019).

[4] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2019. CRYSTALS-Kyber algorithm specifications and supporting documentation. *NIST PQC Round* 2, 4 (2019).

[5] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 4 (2019), 17–61.

[6] M. R. Albrecht, D. J. Bernstein, T. Chou, C. Cid, J. Gilcher, T. Lange, V. Maram, I. von Maurich, R. Misoczki, R. Niederhagen, K. G. Paterson, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, C. J. Tjhai, M. Tomlinson, and W. Wang. 2017. Classic McEliece: Conservative code-based cryptography. *NIST PQC Round* 1 (2017).

[7] Ward Beullens. 2022. Breaking rainbow takes a weekend on a laptop. *IACR Cryptol. ePrint Arch.* (2022), 214. https://eprint.iacr.org/2022/214.

[8] Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology (CRYPTO'97), 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings (1997)*, B. S. K. Jr. (Ed.). Vol. 1294 of Lecture Notes in Computer Science, Springer, 513–525.

[9] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. 2011. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In *International Workshop on Post-quantum Cryptography*. Springer, 117–129.

[10] Johannes Buchmann, Erik Dahmen, and Michael Schneider. 2008. Merkle tree traversal revisited. In *International Workshop on Post-quantum Cryptography*. Springer, 63–78.
[11] Fabio Campos, Tim Kohlstadt, Steffen Reith, and Marc Stöttinger. 2020. LMS vs XMSS: Comparison of stateful hash-based signature schemes on ARM Cortex-M4. *IACR Cryptol. ePrint Arch.* 2020 (2020), 470.
[12] Yuan Cao, Yanze Wu, Wen Wang, Xu Lu, Shuai Chen, Jing Ye, and Chip-Hong Chang. 2022. An efficient full hardware implementation of extended Merkle signature scheme. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 2 (2022), 682–693. https://doi.org/10.1109/TCSI.2021.3115786
[13] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. 2018. Grafting trees: A fault attack against the SPHINCS framework. In *Post-Quantum Cryptography - 9th International Conference (PQCrypto'18), Proceedings (Lecture Notes in Computer Science)*, Tanja Lange and Rainer Steinwandt (Eds.), Vol. 10786. Springer, 165–184. https://doi.org/10.1007/978-3-319-79063-3_8
[14] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. 2019. *Algorithm Specifications and Supporting Documentation*. Brown University and Onboard Security Company, Wilmington USA.
[15] David A. Cooper, Daniel C. Apon, Quynh H. Dang, Michael S. Davidson, Morris J. Dworkin, and Carl A. Miller. 2020. Recommendation for stateful hash-based signature schemes. *NIST Special Publication* 800 (2020), 208.
[16] Jintai Ding and Dieter Schmidt. 2005. Rainbow, a new multivariable polynomial signature scheme. In *International Conference on Applied Cryptography and Network Security*. Springer, 164–175.
[17] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2019. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation. *NIST PQC Round* 2 (2019).
[18] Léo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2017. Crystals-Dilithium: Digital signatures from module lattices. *IACR Cryptol. ePrint Arch.* 633 (2017).
[19] E. Eaton. 2017. Leighton-Micali hash-based signatures in the quantum random-oracle model. In *Selected Areas in Cryptography - SAC'17 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers (2017)*, C. Adams and J. Camenisch, (Eds.). Vol. 10719 of Lecture Notes in Computer Science, Springer, 263–280.
[20] ETSI TC CYBER WG QSC. 2020. Stateful hash-based signatures - public comments on draft SP 800-208. *NIST* (2020).
[21] Ahmed Ferozpuri and Kris Gaj. 2018. High-speed FPGA implementation of the NIST round 1 rainbow signature scheme. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig'18)*. IEEE, 1–8.
[22] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2018. Falcon: Fast-fourier lattice-based compact signatures over NTRU. *Submission to the NIST's PQC Standardization Process* (2018).
[23] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC'96)*. https://doi.org/10.1145/237814.237866
[24] Andreas Hülsing. 2013. W-OTS+ - Shorter signatures for hash-based signature schemes. In *Progress in Cryptology - (AFRICACRYPT'13), 6th International Conference on Cryptology in Africa, Proceedings (Lecture Notes in Computer Science)*, Amr M. Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien (Eds.), Vol. 7918. Springer, 173–188. https://doi.org/10.1007/978-3-642-38553-7_10
[25] Andreas Hülsing, Denis Butin, Stefan Gazdag, Joost Rijneveld, and Aziz Mohaisen. 2018. XMSS: eXtended Merkle signature scheme. *Internet Research Task Force (IRTF), RFC* 8391 (2018), 1–74.
[26] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. 2003. Fractal Merkle tree representation and traversal. In *Cryptographers' Track at the RSA Conference*. Springer, 314–326.
[27] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Hutchinson, A. Jalali, K. Karabina, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik. 2019. SIKE: Supersingular isogeny key encapsulation. *NIST PQC Round* 2 (2019).
[28] Panos Kampanakis and Scott Fluhrer. 2017. LMS vs XMSS: Comparion of two hash-based signature standards. *IACR Cryptology ePrint Archive: Report 2017/349* (2017).
[29] Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. 2018. Differential power analysis of XMSS and SPHINCS. In *International Workshop on Constructive Side-channel Analysis and Secure Design*. Springer, 168–188.
[30] Jonathan Katz. 2016. Analysis of a proposed hash-based signature standard. In *Security Standardisation Research - 3rd International Conference (SSR'16), Proceedings (Lecture Notes in Computer Science)*, Lidong Chen, David A. McGrew, and Chris J. Mitchell (Eds.), Vol. 10074. Springer, 261–273. https://doi.org/10.1007/978-3-319-49100-4_12
[31] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology (CRYPTO'96), 16th Annual International Cryptology Conference, Proceedings (Lecture Notes in Computer Science)*, Neal Koblitz (Ed.), Vol. 1109. Springer, 104–113. https://doi.org/10.1007/3-540-68697-5_9
[32] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Advances in Cryptology (CRYPTO'99), 19th Annual International Cryptology Conference, Proceedings (Lecture Notes in Computer Science)*, Michael J. Wiener (Ed.), Vol. 1666. Springer, 388–397. https://doi.org/10.1007/3-540-48405-1_25

[33] Georg Land, Pascal Sasdrich, and Tim Güneysu. 2021. A hard crystal - Implementing dilithium on reconfigurable hardware. In *Smart Card Research and Advanced Applications - 20th International Conference (CARDIS'21), Revised Selected Papers (Lecture Notes in Computer Science)*, Vincent Grosso and Thomas Pöppelmann (Eds.), Vol. 13173. Springer, 210–230. https://doi.org/10.1007/978-3-030-97348-3_12

[34] David McGrew, Michael Curcio, and Scott Fluhrer. 2019. Leighton-Micali hash-based signatures. *Internet Research Task Force (IRTF), RFC* 8544 (2019).

[35] Ralph C. Merkle. 1989. A certified digital signature. In *Conference on the Theory and Application of Cryptology*. Springer, 218–238.

[36] Prashanth Mohan, Wen Wang, Bernhard Jungk, Ruben Niederhagen, Jakub Szefer, and Ken Mai. 2020. ASIC accelerator in 28 nm for the post-quantum digital signature scheme XMSS. In *2020 IEEE 38th International Conference on Computer Design (ICCD'20)*. IEEE, 656–662.

[37] Dustin Moody. 2019. Round 2 of NIST PQC competition. *PQCrypto* (2019).

[38] D. T. Nguyen, V. B. Dang, and K. Gaj. 2019. A high-level synthesis approach to the software/hardware codesign of NTT-based post-quantum cryptography algorithms. In *2019 International Conference on Field-programmable Technology (ICFPT'19)*. 371–374. https://doi.org/10.1109/ICFPT47387.2019.00070

[39] Lucas Pandolfo Perin, Gustavo Zambonin, Douglas Marcelino Beppler Martins, Ricardo Felipe Custódio, and Jean Everson Martina. 2018. Tuning the Winternitz hash-based digital signature scheme. In *2018 IEEE Symposium on Computers and Communications (ISCC'18)*. IEEE, 537–542. https://doi.org/10.1109/ISCC.2018.8538642

[40] QuantumRISC. 2020. QuantumRISC — Next Generation Cryptography for Embedded Systems. https://www.quantumrisc.org/.

[41] Peter W. Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.* 41, 2 (1999), 303–332. https://doi.org/10.1137/S0036144598347011

[42] Yifeng Song, Xiao Hu, Wenhao Wang, Jing Tian, and Zhongfeng Wang. 2021. High-speed and scalable FPGA implementation of the key generation for the Leighton-Micali signature protocol. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS'21)*. 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401177

[43] Michael Szydlo. 2004. Merkle tree traversal in log space and time. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 541–554.

[44] Jan Philipp Thoma and Tim Güneysu. 2021. A configurable hardware implementation of XMSS. *Cryptology ePrint Archive* (2021).

[45] Ir Frederik Vercauteren, Sujoy Sinha Roy, Jan-Pieter D'Anvers, and Angshuman Karmakar. [n.d.]. SABER: Mod-LWR based KEM.

[46] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. 2019. XMSS and embedded systems. In *International Conference on Selected Areas in Cryptography*. Springer, 523–550.

[47] Kacper Zujko. 2020. Improving differential power analysis of XMSS. In *The Book of Articles National Scientific Conference "Science and Young Researchers" IV edition*. 112.

Received 8 May 2022; revised 25 July 2022; accepted 2 October 2022