XMSS and Embedded Systems
XMSS Hardware Accelerators for RISC-V

Wen Wang[1], Bernhard Jungk[2], Julian Wälde[3], Shuwen Deng[1], Naina Gupta[4], Jakub Szefer[1], and Ruben Niederhagen[3]

[1] Yale University, New Haven, CT, USA
{wen.wang.ww349, shuwen.deng, jakub.szefer}@yale.edu
[2] Independent Researcher
bernhard@projectstarfire.de
[3] Fraunhofer SIT, Darmstadt, Germany
ruben@polycephaly.org
[4] Fraunhofer Singapore, Singapore
naina.gupta@fraunhofer.sg

Abstract. We describe a software-hardware co-design for the hash-based post-quantum signature scheme XMSS on a RISC-V embedded processor. We provide software optimizations for the XMSS reference implementation for SHA-256 parameter sets and several hardware accelerators that allow to balance area usage and performance based on individual needs. By integrating our hardware accelerators into the RISC-V processor, the version with the best time-area product generates a key pair (that can be used to generate 2^{10} signatures) in 3.44 s achieving an over 54× speedup in wall-clock time compared to the pure software version. For such a key pair, signature generation takes less than 10 ms and verification takes less than 6 ms, bringing speedups of over 42× and 17× respectively. This shows that embedded systems equipped with scheme-specific hardware accelerators are able to practically use XMSS. We tested and measured the cycle count of our implementation on an Intel Cyclone V SoC FPGA. The integration of our XMSS accelerators into an embedded RISC-V processor shows that it is possible to use hash-based post-quantum signatures for a large variety of embedded applications.

Keywords: XMSS · Hash-based signatures · Post-quantum cryptography · Hardware accelerator · FPGA · RISC-V

1 Introduction

Due to the continued computerization and automation of our society, more and more systems from consumer products and Internet-of-Things (IoT) devices to cars, high-speed trains and even nuclear power plants are controlled by embedded computers that often are connected to the Internet. Such devices can have a severe impact not only on our information security but increasingly also on our physical safety. Therefore, embedded devices must provide a high level of protection against cyber attacks — despite their typically restricted computing resources. If an attacker is able to disrupt the authenticity of transmitted data, he or she can undermine security of the system in many ways, e.g., malicious firmware can be loaded or contents of a digital document can be changed without being detected. Authenticity of the data is commonly ensured using digital signature schemes, often based on the DSA and ECDSA algorithms [21].

Such currently used asymmetric cryptographic algorithms, however, are vulnerable to attacks using quantum computers: Shor's algorithm [23,24] is able to factor integers and compute discrete logarithms in polynomial time and Grover's algorithm [9] provides a quadratic speedup for brute-force search. In light of recent advances in quantum-computer development and increased research interest in bringing practical quantum computers to life, a new field of post-quantum cryptography (PQC) has evolved [3], which provides cryptographic algorithms that are believed to be secure against attacks using quantum computers. Among these PQC algorithms are a number of algorithms for signing (and verification) of data. This paper focuses on one of these algorithms, the eXtended Merkle Signature Scheme (XMSS), which has been standardized by the IETF [14].

XMSS is a stateful hash-based signature scheme proposed in 2011 by Buchmann, Dahmen and Hülsing [5]. It is based on the Merkle signature scheme [18] and proven to be a forward-secure post-quantum signature scheme with minimal security assumptions: Its security is solely based on the existence of a second pre-image resistant hash function family and a pseudorandom function (PRF) family. Both of these function families can be efficiently constructed even in the presence of large quantum computers [5]. Therefore, XMSS is considered to be a practical post-quantum signature scheme. Due to its minimal security assumptions and its well understood security properties, XMSS is regarded as one of the most confidence-inspiring post-quantum signature schemes.

Embedded devices will need to use algorithms such as XMSS to make them future-proof and to ensure their security even in the light of practical quantum computers. One of the increasingly popular processor architectures for embedded devices is the RISC-V architecture. It is an open and free architecture that is proving to be a practical alternative to close-source designs. Consequently, this work uses a RISC-V-based system on chip (SoC) (see Section 3) as a representative for embedded system architectures and shows how to efficiently deploy the post-quantum signature scheme XMSS on an embedded device, with the help of new hardware accelerators.

Hash-based signature schemes such as XMSS have relatively high resource requirements. They need to perform thousands of hash-computations for key generation, signing and verification and need sufficient memory for their relatively large signatures. Therefore, running such post-quantum secure signature schemes efficiently on a resource-constrained embedded system is a difficult task. This work introduces a number of hardware accelerators that provide a good time-area trade-off for implementing XMSS on RISC-V.

1.1 Our Contributions

Our work is the first that describes a software-hardware co-design of XMSS achieving 128 bit post-quantum security in an embedded systems setting[5]. Profiling results of the XMSS reference software implementation show that around 90% of the time is spent performing the SHA-256 computations. To speed up the XMSS computations, we first provide SHA-256-specific software optimizations for the XMSS reference implementation. Based on the optimized XMSS software implementation, we then develop several hardware accelerators to speed up the most compute-intensive operations in XMSS. Then we integrate the dedicated hardware accelerators into the RISC-V processor and present a software-hardware co-design of XMSS on a RISC-V-based embedded system. Our experimental results show a significant speedup of running XMSS on our software-hardware co-design compared to the pure reference software version. Finally, our work is open-source and can be used and extended freely in research and industry.

Software Optimizations. We propose two software optimizations targeting the most frequently used SHA-256 function in XMSS. First, for most of the hash function calls, the length of the input data is fixed and known before the function call. Therefore, the padding for the input data can be hardcoded given the known input data length. This optimization eliminates the cycles needed for run-time padding computations. The second optimization is based on the fact that for many hash function calls in XMSS, the first block of the input data is fixed. Therefore, the result after the hash computations on the first block is always the same for these hash function calls. This leads to our optimization to store this fixed result after the first hash function call, to reload the stored result for the following calls, and then to finish computations on the rest of the input data. These two software optimizations together bring an over $1.5\times$ speedup to the XMSS reference software implementation.

Hardware Accelerators. Based on the optimized XMSS software implementation, we develop several hardware accelerators to speed up the most expensive operations in XMSS. We first integrate a general-purpose SHA-256 accelerator into the RISC-V processor. Then we develop an XMSS-specific SHA-256 accelerator that adapts the two software optimizations proposed for the XMSS software implementation to hardware. This XMSS-specific SHA-256 accelerator is then used as a building block for two more accelerators each accelerating larger parts of the XMSS computations. These hardware accelerators achieve a significant speedup compared to running the corresponding functions in the optimized XMSS reference implementation in software.

Software-Hardware Co-Design of XMSS. We develop a software-hardware co-design of XMSS on a RISC-V embedded processor, running some parts of the

[5] Intial version of our work was submitted to the Cryptology ePrint Archive as `https://eprint.iacr.org/2018/1225` on 2018/12/21.

computations in software and using hardware accelerators for computationally intensive parts of the XMSS algorithms. Our experimental results with a tree height $h = 10$ (see Section 2.1 for details) on the RISC-V embedded processor (the version with the best time-area product) with a performance of 3.44 s for key generation and around 10 ms and 6 ms respectively for signing and verification show that XMSS is usable even on resource-restricted embedded platforms using hardware acceleration. Our software-hardware co-design can be integrated into other hardware architectures with small adaptations as well.

Open Source. We will release our software-hardware co-design under an open source license to enable academia and industry to fully exploit the benefits of our work. The SHA-256-specific software optimizations, hardware accelerators as well as the software-hardware co-design architectures can be applied to accelerate other cryptosystems whenever applicable.

2 Preliminaries

In this section, we give an introduction to the relevant aspects of the XMSS signature scheme and briefly recapitulate the functionalities of SHA-256.

2.1 XMSS

The eXtended Merkle Signature Scheme (XMSS) [14] is a stateful signature scheme based on the Merkle signature scheme [18]. Similar to the Merkle signature scheme, XMSS uses the Winternitz one-time signature scheme (WOTS or Winternitz-OTS) to sign individual messages [18]. One private/public WOTS key pair is used to sign one single message (with the private secret key) and to verify the signature (with the corresponding public verification key). To be able to sign up to 2^h messages, XMSS uses 2^h pairs of WOTS secret and verification keys. To reduce the size of the public key, a Merkle hash tree of height h and binary L-trees are used to reduce the authenticity of many WOTS verification keys to one XMSS public key. Since each WOTS key must only be used once, the signer needs to remember which WOTS keys already have been used. Hence, the scheme is stateful. Figure 1 shows the overall structure of XMSS.

The XMSS standard also defines multi-tree versions called XMSSˆMT where the leaf nodes of a higher-level tree are used to sign the root of another tree. In this paper, we mainly consider single-tree XMSS. However, our results apply to multi-tree XMSS as well in a straightforward way. For a detailed description of XMSS and XMSSˆMT please refer to IETF RFC 8391 [14] and to [5].

In the following we briefly introduce the XMSS address scheme, WOTS, the L-tree construction, and the procedure for constructing the Merkle tree. We also give an introduction to XMSS key generation, signing, and verification.

Address Scheme. XMSS uses a hash function address scheme throughout the Merkle tree, L-tree, and WOTS computations to uniquely identify each individual step in the overall graph. These addresses are used to derive keys for keyed

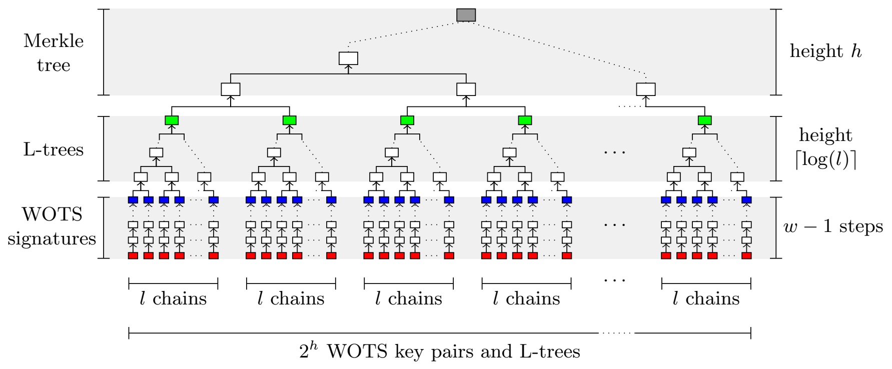

Fig. 1: XMSS tree with binary Merkle hash tree and WOTS instances with L-trees as leaves. Red nodes are the WOTS private key and blue nodes are the WOTS public key values. Green nodes are the L-tree roots and the gray node is the XMSS public key.

hash functions that are unique for each specific location in the graph. Each address is composed of eight 32 bit fields, with fields for, e.g., the level within a tree and the leaf index. In total, an XMSS address has a size of 256 bit. For more details about the hash function address scheme, please refer to IETF RFC 8391 [14, Sect. 2.5].

Winternitz OTS. The WOTS scheme was first mentioned in [18]. For signing a message digest D of n-byte length, WOTS uses a cryptographically secure hash function with n-byte output strings to compute hash chains. The message digest is interpreted as binary representation of an integer d. First, d is split into $l_1 = \lceil 8n/\log_2(w) \rceil$ base-w words $d_i, 0 \leq i < l_1$ and a checksum $c = \sum_{i=0}^{l_1} w - 1 - d_i$ is computed for these base-w words (w is called the "Winternitz parameter"). The checksum c is split into $l_2 = \lfloor \log_2(l_1(w-1))/\log_2(w)) + 1 \rfloor$ base-w words $c_i, 0 \leq i < l_2$. WOTS key generation, signing, and verification are performed as follows:

- To *create* a private/public WOTS key pair, Alice computes $l = l_1 + l_2$ secret strings $s_{0,0}, s_{0,1}, \ldots, s_{0,l-1}$, each of n-byte length (for example using a secret seed and a PRF). These l n-byte strings are the private WOTS key. Then, Alice uses a chaining function to compute l hash chains of length $w-1$, hashing each $s_{0,i}$ iteratively $w-1$ times. The resulting chain-head values $s_{w-1,i}, 0 \leq i < l$ of n-byte length are the public WOTS key and are published by Alice.
- To *sign* a message digest D split into l_1 base-w words together with l_2 base-w checksum values computed as described above, Alice (re-)computes the intermediate chain values $(s_{d_0,0}, s_{d_1,1}, \ldots, s_{d_{l_1-1},l_1-1}, s_{c_0,0}, s_{c_1,1}, \ldots, s_{c_{l_2-1},l_2-1})$ starting from her private key values. These $l = l_1 + l_2$ values are the signature.
- When Bob wants to *verify* the signature, he recomputes the remaining chain steps by applying, e.g., $w-1-d_0$ hash-function iterations to signature value $s_{d_0,0}$ and compares the resulting values with the corresponding public key

values. If all chain-head values match the public WOTS key, the signature is valid.

XMSS uses a modified WOTS scheme, sometimes referred to as WOTS+ or as W-OTS+ [12]; we use the term WOTS+ only when a explicit distinction from "original" WOTS is required for clarification. WOTS+ uses a function chain() as chaining function that is a bit more expensive than the simple hash-chain function described above. The function chain() uses a keyed pseudo-random function $\mathrm{prf}_k : \{0,1\}^{256} \mapsto \{0,1\}^{8n}$ and a keyed hash-function $\mathrm{f}_{k'} : \{0,1\}^{8n} \mapsto \{0,1\}^{8n}$. Within each chain step, the function chain() first computes a unique n-byte key k' and a unique n-byte mask using the $\mathrm{prf}_k()$ function. The input to $\mathrm{prf}_k()$ is the hash function address of the current step (including the chain step and a marker for the usage as key or as mask). The key k for $\mathrm{prf}_k()$ is a seed that is part of the XMSS public key. The mask is then XOR-ed with the n-byte output from the previous chain-function call (or the initial WOTS+ chain n byte input string) and the result is used as input for the hash-function f() under the key k', which gives the n-byte output of the chaining function chain() in the last iteration step.

The WOTS+ secret key consists of l (l is defined as described above for WOTS) pseudo-random strings of n-bytes in length. The XMSS specification does not demand a certain function to compute the WOTS+ private key. In the XMSS reference implementation, they are generated using the $\mathrm{prf}_k()$ function with the local address (including the chain index) as input and keyed with the XMSS secret key seed. Each WOTS+ secret key maps to one corresponding WOTS+ public key, which is computed by calling the chaining function chain() with $w - 1$ iteration steps. Signing and verification in WOTS+ work as described above for WOTS using the WOTS+ chaining function. The more complex structure of the chaining function of WOTS+ compared to WOTS is required for multi-target resistance and within the XMSS security proof.

L-tree. The leaf nodes of an XMSS tree are computed from the WOTS+ public keys by using an unbalanced binary tree of l leaf nodes (one leaf node for each WOTS+ public key value), hence called L-tree. The nodes on each level of the L-tree are computed by hashing together two nodes from the lower level. A tree hash function $\mathrm{hash}_{\mathrm{rand}} : \{0,1\}^{8n} \times \{0,1\}^{8n} \mapsto \{0,1\}^{8n}$ is used for this purpose.

The function $\mathrm{hash}_{\mathrm{rand}}()$ uses the keyed pseudo-random function $\mathrm{prf}_k()$ and a keyed hash-function $\mathrm{h}_{k''} : \{0,1\}^{16n} \mapsto \{0,1\}^{8n}$. First, an n-byte key k'' and two n-byte masks are computed using the $\mathrm{prf}_k()$ with the address (including the L-tree level and node index) as input and the same public seed as used for WOTS+ as key. The masks are then each XOR-ed to the two n-byte input strings representing the two lower-level nodes and the results are concatenated and used as input for the hash-function h() keyed with k'', which gives the n-byte output of the tree hash function $\mathrm{hash}_{\mathrm{rand}}()$.

To be able to handle the pairwise hashing at levels with an odd number of nodes, the last node on these levels is lifted to a higher level until another single node is available. The root of the L-tree gives one single hash-value, combining the l WOTS+ public keys into one WOTS+ verification key.

XMSS Merkle Tree. In order to obtain a small public key, the authenticity of many WOTS verification keys (i.e., L-tree root keys) is reduced to one XMSS public key using a binary Merkle tree. Similar to the L-tree construction described above, on each level of the binary tree, neighbouring nodes are pairwise hashed together using the $\text{hash}_{\text{rand}}()$ function to obtain one single root node that constitutes the XMSS public key (see Figure 1).

XMSS Key Generation. XMSS key generation is quite expensive: In order to compute the XMSS public key, i.e., the root node of the Merkle tree, the entire XMSS tree needs to be computed. Depending on the height h of the tree, thousands to millions of hash-function calls need to be performed. XMSS key generation starts by generating 2^h leaf nodes of the Merkle tree. Each leaf node consists of an WOTS instance together with an L-tree. For each WOTS instance, first l WOTS private keys are generated. These are then used to compute the l WOTS chains to obtain l WOTS public keys and then the L-trees on top of these. Once all 2^h L-tree root nodes have been computed, the Merkle tree is computed to obtain the XMSS public key.

The XMSS public key consists of the n-byte Merkle tree root node and the n-byte public seed required by the verifier to compute masks and public hash-function keys using the function $\text{prf}_k()$ within the WOTS-chain, L-tree, and Merkle tree computations. The XMSS standard does not define a format for the XMSS private key. In the XMSS reference implementation that accompanies the standard, an n-byte secret seed is used to generate the WOTS secrets using a pseudo random function (e.g., $\text{prf}_k()$).

XMSS Signature Generation. XMSS is a stateful signature scheme: Each WOTS private/public key pair must be used only once; otherwise, the scheme is not secure. To determine which WOTS key pair already has been used, an n-byte leaf index (the state) is stored with the private key. The index defines which WOTS key pair will be used for the next signature; after each signature generation, the index must be increased.

Similar to most signature schemes, for signing an arbitrary-length message or a document M, first a message digest of M is computed; details can be found in [14, Sect. 4.1.9]. The digest M' is then signed using the selected WOTS instance. This results in l n-byte values corresponding to the base-w decomposition of M' including the corresponding checksum. Furthermore, in order to enable the verifier to recompute the XMSS public root key from a leaf node of the Merkle tree, the signer needs to provide the verification path in the Merkle tree, i.e., h n-byte nodes that are required for the pairwise hashing in the binary Merkle tree, one node for each level in the Merkle tree.

Therefore, in the worst case, the signer needs to recompute the entire XMSS tree in order to select the required nodes for the verification path. There are several optimization strategies using time-memory trade-offs to speed up signature generation. For example, the signer can store all nodes of the Merkle tree up to level h' alongside the private key. Then, when signing, the signer only needs to compute an $(h - h')$-height sub-tree including the WOTS leaves and can reproduce the signature path for the remaining h' levels from the stored data. Other algorithms with different trade-offs exist; for example the BDS tree traversal algorithm [6] targets at reducing the worst case runtime of signature generation by computing a certain amount of nodes in the Merkle tree at each signature computation and storing them alongside the XMSS state.

XMSS Signature Verification. Compared to key generation, XMSS signature verification is fairly inexpensive: An XMSS public key contains the Merkle root node and the public seed. An XMSS signature contains the WOTS leaf index, l WOTS-signature chain values, and the verification path consisting of h Merkle-tree pair values, one for each level in the tree. The verifier computes the message digest M' and then recomputes the WOTS verification key by completing the WOTS chains and computing the L-tree. The verifier then uses the Merkle-tree pair values to compute the path through the Merkle tree and finally compares the Merkle tree root node that was obtained with the root node of the sender's public key. If the values are equal, verification succeeds and the signature is sound; otherwise verification fails and the signature is rejected.

Parameter Set. RFC 8391 defines parameter sets for the hash functions SHA-2 and SHAKE targeting classical security levels of 256 bit with $n = 32$ and 512 bit with $n = 64$ in order to provide 128 bit and 256 bit of security respectively against attackers in possession of a quantum computer [14, Sect. 5]. The required parameter sets, as specified in [14, Sect. 5.2], all use SHA-256 to instantiate the hash functions. Therefore, for this work, we focus on the SHA-256 parameter sets with $n = 32$.

In this case, the keyed hash functions $\mathrm{prf}_k : \{0,1\}^{256} \mapsto \{0,1\}^{256}$, $\mathrm{f}_{k'} : \{0,1\}^{256} \mapsto \{0,1\}^{256}$, and $\mathrm{h}_{k''} : \{0,1\}^{512} \mapsto \{0,1\}^{256}$, are implemented by computing the input to SHA-256 as concatenation of:

– a 256 bit hash-function specific domain-separator,

– the 256 bit hash-function key, and

– the 256 bit or 512 bit hash-function input.

For SHA-256, three different parameter sets are provided in RFC 8391 [14, Sect. 5.3], all with $n = 32$ and $w = 16$ but with $h = 10$, $h = 16$, or $h = 20$. In general, a bigger tree height h leads to an exponential growth in the run time of key generation. For verification the run time is only linearly impacted. The naive approach for signing requires one to recompute the entire tree and thus is as expensive as key generation. However, by use of the BDS tree traversal algorithm [6], the tree height has only a modest impact on the run time. Multi-tree versions of XMSS (XMSSˆMT) can be used to speed up the computations at the cost of larger signature sizes (e.g., to improve key generation and signing performance or to achieve a larger h). We are using $h = 10$ throughout our experiments; however, our implementation is not restricted to this value.

2.2 SHA-256

The hash function SHA-256 [20] computes a 256 bit hash value from a variable-length input. SHA-256 uses a 256 bit internal state that is updated with 512 bit blocks of the input. Therefore, SHA-256 defines a padding scheme for extending variable-length inputs to a multiple of 512 bit. SHA-256 works as follows:

1. Initialize the internal state with a well-defined IV (see [20, Sect. 4.2.2]).
2. Extend the ℓ-bit input message with a padding to make the length of the padded input a multiple of 512 bit:
   - append a single 1 bit to the input message, then
   - append $0 \leq k$ 0 bit such that $\ell + 1 + k + 64$ is minimized and is a multiple of 512, and finally
   - append ℓ as a 64 bit big-endian integer.
3. Iteratively apply a compression function to all 512 bit blocks of the padded input and the current internal state to obtain the next updated internal state.
4. Once all 512 bit blocks have been processed, output the current internal state as the hash value.

The compression function uses the current internal state and a 512 bit input block and outputs an updated internal state. For SHA-256, the compression function is composed of 64 rounds.

3 RISC-V

Software-hardware co-design has been adopted as a common discipline for designing embedded system architectures since the 1990s [26]. By combining both software and hardware in an embedded system, a trade-off between software flexibility and hardware performance can be achieved depending on the user's needs. To accelerate XMSS computations, we developed a software-hardware co-design of XMSS by moving the most compute-intensive operations to hardware while keeping the rest of the operations running in software. Our software-hardware co-design of XMSS is developed based on a RISC-V platform.

RISC-V. The RISC-V instruction set architecture (ISA) is a free and open architecture, overseen by the RISC-V Foundation with more than 100 member organizations[6]. The RISC-V ISA has been designed based on well-established reduced instruction set computing (RISC) principles. It has a modular design, consisting of base sets of instructions with optional instruction set extensions. Due to its modular design, the RISC-V ISA is an increasingly popular architecture for embedded systems. It is used, e.g., as a control processor in GPUs and in storage devices [10], for secure boot and as USB security dongle [19], and for

[6] `https://riscv.org/`

building trusted execution environments (TEE) with secure hardware enclaves[7]. Since the RISC-V ISA is an open standard, researchers and industry can easily extend and adopt it in their designs without IP constraints.

VexRiscv. First-prize winner in the RISC-V Soft-Core CPU Contest of 2018[8], VexRiscv[9] is a 32-bit RISC-V CPU implementation written in SpinalHDL[10], which is a Scala-based high-level hardware description language. It supports the RV32IM instruction set and implements a 5-stage in-order pipeline. The design of VexRiscv is very modular: All complementary and optional components are implemented as plugins and therefore can easily be integrated and adapted into specific processor setups as needed. The VexRiscv ecosystem provides memories, caches, IO peripherals, and buses, which can be optionally chosen and combined as required.

Murax SoC. The VexRiscv ecosystem also provides a complete predefined processor setup called "Murax SoC" that has a compact and simple design and aims at small resource usage. The Murax SoC integrates the VexRiscv CPU with a shared on-chip instruction and data memory, an Advanced Peripheral Bus (APB), a JTAG programming interface, and a UART interface. It has very low resource requirements (e.g., only 1350 ALMs on a Cyclone V FPGA) and can operate on its own without any further external components.

The performance of the Murax SoC is comparable to an ARM Cortex-M3: A multi-tree version of XMSS has been implemented on an embedded ARM Cortex-M3 platform in [15] (see also Section 7.3). We compiled a pure C-version of the code from [15] for both an ARM Cortex-M3 processor and the Murax SoC and compared their performance. In terms of cycle count, the Cortex-M3 is only about 1.5× faster than the Murax SoC. Therefore, we conclude that the Murax SoC is a good representative for an embedded system processor with low resources. As opposed to an ARM Cortex-M3 platform, however, the Murax SoC is fully free, open, and customizable and thus is an ideal platform for our work.

Extending the Murax SoC with new hardware accelerators can be implemented easily in a modular way using the APB bus. We used this feature for our XMSS accelerators. Depending on different use cases, our open-source software-hardware co-design of XMSS can be migrated to other RISC-V or embedded architectures with small changes to the interface.

Setup. We evaluated our design using a DE1-SoC evaluation board from Terasic as test-platform. This board has an Intel (formerly Altera) Cyclone V SoC 5CSEMA5F31C6 device with about 32,000 adaptive logic modules (ALMs) and about 500 KB of on-chip memory resources. (We do not use the DSP resources or the ARM Cortex-A9 CPU of the device.) We used Intel Quartus Software

[7] `https://keystone-enclave.org/`
[8] `https://riscv.org/2018/10/risc-v-contest/`
[9] `https://github.com/SpinalHDL/VexRiscv/`
[10] `https://spinalhdl.github.io/SpinalDoc/`

Version 16.1 (Standard Edition) for synthesis. On the DE1-SoC, we are running the Murax SoC described above with additional accelerators that will be described in Section 5. The DE1-SoC board is connected to a host computer by a USB-JTAG connection for programming the FPGA, a USB-serial connection for IO of the Murax SoC, and a second USB-JTAG connection for programming and debugging the software on the Murax SoC.

We configured the on-chip RAM size of the Murax SoC to 128 kB, which is sufficient for all our experiments. We tested our implementations on the DE1-SoC board at its default clock frequency of 50 MHz; however, to achieve a fair comparison, our speedup reports presented in the following sections are based on the maximum frequency reported by the synthesis tool.

Our implementation is neither platform-specific nor dependent on a specific FPGA vendor.

4 Software Implementation and Optimization

We used the official XMSS reference implementation[11] as software-basis for this work. We applied minor modifications to the XMSS reference code to link against the mbed TLS library[12] instead of OpenSSL, because mbed TLS generally is more suitable for resource-restricted embedded platforms such as the Murax SoC platform and its SHA-256 implementation has less library-internal dependencies than that of OpenSSL, which simplifies stand-alone usage of SHA-256.

The tree-hash algorithm [14] used for computing the XMSS public key and the authentication path within the Merkle tree requires an exponential number of 2^h WOTS operations for computing tree leaves. However, key generation and signing are not memory intensive when the tree is computed with a depth-first strategy. The XMSS reference implementation provides two algorithms for signature generation. The first approach (implemented in file "xmss_core.c") straightforwardly re-computes all tree leaf nodes in order to compute the signature authentication path and therefore has essentially the same cost as key-generation. This approach does not require to store any further information. The second approach (implemented in file "xmss_core_fast.c") uses the BDS algorithm [6] to make a trade-off between computational and memory complexity. It requires to additionally store a state along the private key. Both versions can be used with our hardware accelerators. Our experiments show that both versions of the signature generation algorithm run smoothly on the Murax SoC. Even with the additional storage requirement, running all the operations of XMSS with the BDS-based signature algorithm leads to reasonable memory usage, as shown in Section 6. Since the runtime of the basic signature algorithm is almost identical to key generation, we are using the fast BDS version of the signature algorithm [6] for our performance reports.

To have a fair reference point for the comparison of a pure software implementation with our hardware accelerators, we implemented two software opti-

[11] `https://github.com/joostrijneveld/xmss-reference/`, commit 06281e057d9f5d
[12] `https://tls.mbed.org/`

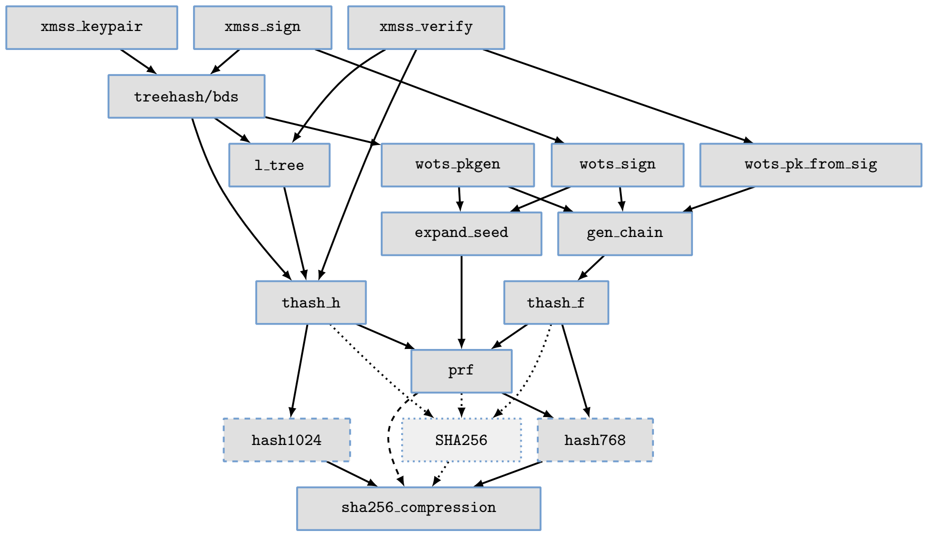

Fig. 2: Simplified XMSS call graph. Function calls that have been removed during software optimization(i.e., calls to SHA256 including init, update, and finish) are displayed with dotted nodes and arrows. Added calls for the "fixed input length" optimization are displayed as dashed nodes and for the "pre-computation" optimization are displayed as dashed arrows.

mizations for the XMSS reference software implementation as described in the following text. These optimizations are also helpful on other processor architectures but only work for SHA-256 parameter sets, because they depend on the specific SHA-256 block size and padding scheme. We are going to provide our software optimizations to the maintainers of the XMSS reference implementation so they can integrate them if they wish to.

Figure 2 shows a simplified XMSS call graph for both the original source code version and the changes that we applied for optimizations as described below (dotted nodes and edges have been removed and dashed nodes and edges haven been added during optimization).

4.1 Fixed Input Length

In the XMSS software reference implementation, around 90% of the time is spent inside the hash-function calls. Therefore, the SHA-256 function is most promising for optimization efforts. The main interface to SHA-256 in mbed TLS has three functions, `mbedtls_sha256_init`, `mbedtls_sha256_update`, and `mbedtls_sha256_finish` (combined and simplified to `SHA256` in Figure 2). The "init"-function initializes the internal state of the SHA-256 implementation. The "update"-function allows to feed in message chunks of arbitrary size and updates the internal state accordingly. The "finish" function finally adds the padding and returns the message digest. Internally, these functions need to adapt arbitrary-length message chunks to the SHA-256 input block size of 512 bit: If the size of message-chunk input to the update function `mbedtls_sha256_update` is not a multiple of 512 bit, the remaining data is buffered alongside the internal state and used either in the next "update" or in the final "finish" call.

The SHA-256 implementation of mbed TLS is intended to hash messages of an arbitrary length: When the "finish" function is called, the actual length of the entire message is computed as sum over the lengths of all individual message chunks and the padding is generated accordingly. However, within the XMSS scheme, the inputs of almost all SHA-256 calls have a well-known, fixed length: A general, arbitrary-length SHA-256 computation is only required when computing the actual hash digest of the input message, which is called only once for signing and once for verifying. For all the other SHA-256 calls, the length of the input data is either 768 bit or 1024 bit depending on where SHA-256 is called within the XMSS scheme: An input length of 768 bit is required within the PRF and within the WOTS-chain computation; an input length of 1024 bit is required within the Merkle tree and the L-trees to hash two nodes together. Therefore, we can eliminate the overhead for the padding computation of the SHA-256 function by "hardcoding" the two required message paddings, given that their lengths are known beforehand.

Implementation. We implemented two specialized SHA-256 functions: The function `hash768` targeting messages with a fixed length of 768 bit and function `hash1024` targeting messages with fixed length of 1024 bit. Figure 3 shows the padding for `hash768` and `hash1024`. Since SHA-256 has a block size of 512 bit, two blocks are required to hash a message of length 768 bit. Therefore, we need to hardcode a 256 bit padding for `hash768` to fill up the second block to 512 bit. When a 768 bit message is fed to the `hash768` function, the 256 bit padding is appended to the message. Then, the new 1024 bit padded message is divided into two 512 bit blocks and the compression function is performed on each of them one by one. Once the compression function on the second message block has finished, the internal state is read out and returned as the output.

The SHA-256 standard always demands to append a padding even if the input length is a multiple of 512 bit. Therefore, for the `hash1024` function a 512 bit padding is hardcoded similarly to `hash768` and three calls to the compression function are performed.

Evaluation. Table 1 shows a comparison of the original XMSS reference implementation with an optimized version making use of the "fixed input length" optimization on the Murax SoC with parameters $n = 32$, $w = 16$ and $h = 10$. The speedup for 768 bit inputs is about 1.07× and for 1024 bit inputs is about 1.04×. The use of 768 bit inputs is more common during the XMSS computations. Therefore, we see an about 1.06× speedup for WOTS computations as well as the key generation, signing, and verification operations in XMSS.

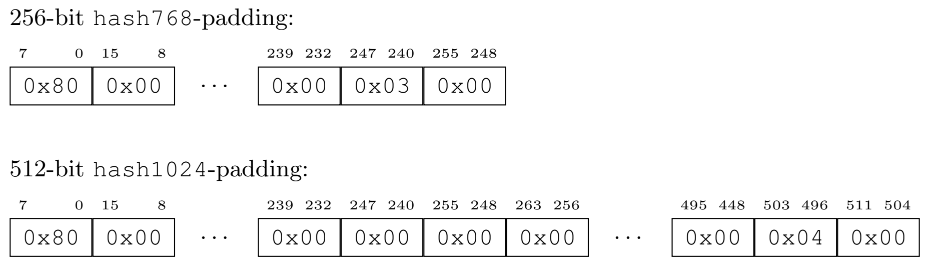

Fig. 3: Fixed padding for `hash768` and `hash1024`
.

4.2 Pre-Computation

Pre-computation is commonly referred to as the act of performing an initial computation before runtime to generate a lookup table to avoid repeated computations during runtime. This technique is useful in improving real-time performance of algorithms at the expense of extra memory and extra preparatory computations [2]. In XMSS, a variant of this idea can be applied to improve the performance of the hash functions.

Within XMSS, SHA-256 is used to implement four different keyed hash-functions, the function `thash_f` for computing f() in the WOTS-chains, the function `thash_h` for h() in the tree hashing, and the function `prf` for computing the prf(), generating masks and hash-function keys. Furthermore, SHA-256 is used to compute the message digest that is signed using a WOTS private key. The domain separation and the keying for these four functions are achieved by computing the input to SHA-256 as the concatenation of a 256 bit domain separator value (distinct for these four functions), the 256 bit hash key, and the hash-function input. Since SHA-256 operates on 512 bit blocks, one entire block is required for domain separation and keying of the respective hash function.

In case of the `prf`, for all public-key operations when generating masks and hash-function keys for the WOTS chain, the L-tree and Merkle tree operations, the key to the `prf` is the 256 bit XMSS public seed. Thus, both the 256 bit domain separator and the 256 bit hash-function key are the same for all these calls for a given XMSS key pair. These two parts fit exactly into one 512 bit SHA-256 block. Therefore, the internal SHA-256 state after processing the first 512 bit block is the same for all these calls to the `prf` and based on this fact, we can save one SHA-256 compression function call per `prf`-call by pre-computing and replaying this internal state. The internal state can either be computed once and stored together with the XMSS key or each time an XMSS operation (key generation, signing, verification) is performed. This saves the computation on one of the two input blocks in `hash768` used in the `prf`. For `hash1024`, this optimization is not applicable since the fixed input block pattern does not exist.

Implementation. At the first call to `prf`, we store the SHA-256 context of mbed TLS for later usage after the first compression function computation. The

	"original"	+ "fixed input length"		+ "pre-computation"		
	Cycles	Cycle	Speedup	Cycles	Speedup	Speedup
	(A)	(B)	(AB)	(C)	(BC)	(AC)
`hash768`	11.5×10^3	10.7×10^3	1.07	5.87×10^3	1.83	1.95
`hash1024`	16.2×10^3	15.6×10^3	1.04	—	—	—
WOTS-chain	571×10^3	530×10^3	1.08	371×10^3	1.43	1.54
WOTS-leaf	42.2×10^6	39.8×10^6	1.06	27.7×10^6	1.44	1.53
key generation	43.3×10^9	40.8×10^9	1.06	28.3×10^9	1.44	1.53
signing	58.3×10^6	55.0×10^6	1.06	38.4×10^6	1.43	1.52
verification	26.7×10^6	25.2×10^6	1.06	17.4×10^6	1.45	1.54

Table 1: Cycle count and speedup of the "fixed input length" optimization and for both, the "fixed input length" and the "pre-computation" optimizations, on the Murax SoC with parameters $n = 32$, $w = 16$ and $h = 10$.

state includes the internal state and further information such as the length of the already processed data.

When the `prf` is called during XMSS operations, we first create a copy of the initially stored `prf` SHA-256 context and then perform the following prf() operations based on this state copy, skipping the first input block. The cost for the compression function call on the first SHA-256 block within the `prf` is therefore reduced to a simple and inexpensive memory-copy operation.

Evaluation. Performance measurements and speedup for our pre-computation optimization are shown in Table 1. For `hash768` we achieve a 1.83× speedup over the "fixed input length" optimization (column "Speedup (BC)"), because only one SHA-256 block needs to be processed instead of two. Compared to the original non-optimized version, with both optimizations (including "fixed input length") enabled we achieve an almost 2× speedup (column "Speedup (AC)").

The function `thash_f` for computing WOTS-chains requires two calls to the `prf` (each on two SHA-256 blocks) for generating a key and a mask and one call to `hash768` (on two SHA-256 blocks). Without pre-computation, six calls to the SHA-256 compression function are required. With a pre-computed initial state for the `prf`, only four calls to the SHA-256 compression function are required, saving one third of the compression function calls. This optimization leads to a 1.43× speedup for WOTS-chain computations (row "WOTS-chain", column "Speedup (BC)"). The overall speedup including both optimizations "pre-computation" and "fixed input length" is 1.54×.

For L-tree computations within the randomized tree-hash function `thash_h`, there are three calls to the `prf` (each on two SHA-256 blocks) for computing two masks and one hash-function key and one call to `hash1024` (on three SHA-256 blocks). Without pre-computation, nine calls to the SHA-256 compression function are required. With a pre-computed initial state for the `prf`, only six calls to the SHA-256 compression function are required, again saving one third

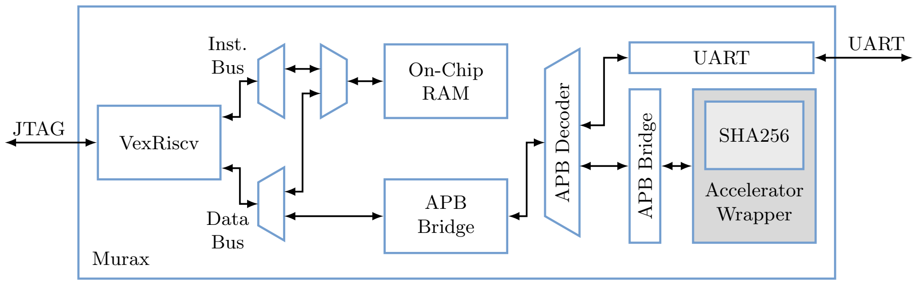

Fig. 4: Schematic of the Murax SoC. Hardware accelerators are connected to the APB. Details on the hardware accelerators are shown in Figure 5.

of the compression function calls. This optimization leads to a 1.44× speedup for the overall XMSS leaf computations (see Table 1, row "WOTS-leaf"). The speedup including both optimizations is around 1.53×.

The expected speedup for Merkle tree computations is about the same as for the L-tree computations since the trees are constructed in a similar way. Table 1 shows that we achieve an overall speedup of more than 1.5× including both optimizations also for the complete XMSS operations, i.e., key generation, signing, and verification. We observed a similar speedup on an Intel i5 CPU.

5 Hardware Acceleration

To further accelerate the XMSS computations, we developed several dedicated hardware modules together with software interfaces for the XMSS software. As shown in Figure 4, the Murax SoC uses an APB for connecting peripherals to the main CPU core. The peripheral can be accessed by the software running on the Murax SoC via control and data registers that are mapped into the address space. Therefore, the software interface can simply use read and write instructions to communicate with a hardware module. Due to the modularity of the VexRiscv implementation, dedicated hardware modules can be easily added to and removed from the APB before synthesis of the SoC (see Section 3).

We developed a general-purpose SHA-256 accelerator for accelerating the compression function of SHA-256 in hardware and the following XMSS-specific hardware accelerators: An XMSS-specific SHA-256 accelerator with fixed-length SHA-256 padding and an optional internal storage for pre-computation, a WOTS-chain accelerator for the WOTS chaining computations, and an XMSS-leaf generation accelerator combining WOTS and L-tree computations.

The hardware accelerators are connected to the APB using a bridge module: The `Apb3Bridge` module connects on one side to the 32 bit data bus and the control signals of the APB and on the other side to the hardware accelerator. It provides a 32 bit control register, which is mapped to the control and state ports of the hardware accelerator, and data registers for buffering the input data, which are directly connected to the input ports of the hardware accelerator.

The control and data registers are mapped to the APB as 32 bit words using a multiplexer, selected by the APB address port on APB write; the control register and the output ports of the hardware accelerator are connected in the same way to be accessed on APB read. This allows the software to communicate with the accelerators via memory-mapped IO using simple load and store instructions.

We modified the corresponding software functions in the optimized XMSS implementation to replace them with function calls to our hardware accelerators as follows: The function first sets control bits (e.g., `RESET`, `INIT`) to high in the control register. When these bits are received as high by the `Apb3Bridge` module, it raises the corresponding input signals of the hardware accelerator. Similarly, the input data is sent to the corresponding hardware accelerator via the APB bus in words of width 32 bit. Then the computation of the hardware accelerator is triggered by setting the `COMP` bit in the control register to high which further toggles the input port `start` of the hardware accelerator. Then the hardware accelerator begins its computation. Once the computation is finished, the hardware accelerator raises the output port `done` and the APB interface sets the `DONE` bit in the control register to high.

Once the software is ready to read the result, it keeps polling the control register until the `DONE` bit is set high. The software then can read the results via the APB in words of 32 bit.

5.1 Baseline: General-Purpose SHA-256 Accelerator

Since around 90% of the time is spent performing the SHA-256 computations in the XMSS reference implementation, the first hardware module we developed is the `SHA256` module, which is a general-purpose hash accelerator that accepts variable length inputs. The `SHA256` module is used as the building block in the XMSS-specific hardware accelerators described in the following sub-sections. It has a similar interface as the generic SHA-256 compression function in software: It receives a 512 bit data block as input and computes the compression function, updating an internal 256 bit state. This state can be read out as the 256 bit digest when the SHA-256 computation is finished. Padding is performed in software as before.

We developed the module `SHA256` by implementing an iterative version of SHA-256 by computing one round of SHA-256 in one clock cycle. Therefore, we require 64 clock cycles to process one SHA-256 data block. This provides a good trade-off between throughput and area consumption similar to [11]. The `SHA256` module is a generic hash core without platform-specific optimizations that runs on any FPGA platform. Users can easily use a platform-optimized SHA-256 core within our hardware modules, e.g., [7,16,22].

Hardware Support for Software Optimizations. The software optimization of SHA-256 exploiting fixed input lengths of the SHA-256 function described in Section 4.1 can be mapped in a straightforward way to the `SHA256` module. The software prepares the SHA-256 input chunks with pre-defined paddings just as before and then transfers each chunk to the `SHA256` module for processing.

Therefore, the speedup achieved for the pure software version can also be exploited for this hardware accelerator.

In order to support the "pre-computation" optimization (Section 4.2), we added an interface to the `SHA256` module that allows to set the internal state of the `SHA256` module from software. Reading the internal state is the same as reading the SHA-256 message digest after the last compression function computation.

We modified the function `mbedtls_sha256_init` from mbed TLS to replace the software implementation of the SHA-256 compression function with a call to our hardware accelerator as follows: The function first sets the `INIT` bit to high in the control register. When this bit is received as high by the `Apb3Bridge` module, it raises the `init_message` signal of the `SHA256` module, which resets the value of internal state register to the SHA-256 initialization value. In order to set the internal state for the pre-computation optimization, the software writes a previously stored state to the data register and then sets the control register bit `LOAD_IV` to high. Once the APB interface sees this bit as high, it sets the `init_iv` signal to high and the `SHA256` module sets the internal state to the 256 least significant bits of the input signal `data_in`. When the compression function is called in software, the 512 bit input message block is sent to the `SHA256` module via the APB bus in words of width 32 bit. Then, the `SHA256` computation is triggered.

While the hardware is performing the hash computation, the software can go on transferring the next data block to the `SHA256` module. This reduces the communication overhead and increases the efficiency of the `SHA256` module.

Evaluation. Table 2 shows performance, resource requirements, and maximum frequency of the `SHA256` module. The module requires 64 cycles (one cycle per round) for computing the compression function on one 512 bit input block.

Table 2 also shows a comparison of computing one SHA-256 compression function call in software (design `Murax`) with calling the hardware module from the software (design "`Murax + SHA256`"). Transferring data to the `SHA256` accelerator module and reading back the results contribute a significant overhead: The entire computation on a 512 bit input block (without SHA-256 padding computation) requires 253 cycles. This overhead is due to the simple bus structure of the Murax SoC; a more sophisticated bus (e.g., an AXI bus) may have a lower overhead — at a higher cost of resources. However, we achieve an almost $13\times$ speedup over the software implementation of the SHA-256 compression function from the mbed TLS library which requires about 4950 cycles on the Murax SoC.

For one regular `hash768` function call, the SHA-256 compression function needs to be performed on two 512 bit blocks, while for one `hash1024` function call, three 512 bit blocks are needed. When the "pre-computation" optimization is enabled in the software, only one 512 bit block needs to be compressed in a `hash768` function call.

Table 6 shows the performance impact of the `SHA256` module on XMSS computations (designs `Murax` and "`Murax + SHA256`", including both "fixed input

Design	Cycles	Area (ALM)	Reg.	Fmax (MHz)	Time (µs)	Time×Area (relative)	Speedup
			one 512 bit block				
`SHA256`	64	1180	1550	100	0.639	—	—
		`hash768` with pre-computation (one 512 bit block)					
`Murax`	4950	1350	1660	152	32.6	9.22	1.00
+ `SHA256`	253	2860	3880	99.9	2.53	1.00	12.9
		`hash768` without pre-computation (two 512 bit blocks)					
`Murax`	10,700	1350	1660	152	70.4	8.76	1.00
+ `SHA256`	576	2860	3880	99.9	5.77	1.00	12.2
		`hash1024` (three 512 bit blocks)					
`Murax`	15,600	1350	1660	152	102	10.5	1.00
+ `SHA256`	700	2860	3880	99.9	7.01	1.00	14.6

Table 2: Performance of the hardware module `SHA256` and comparisons of performing the SHA-256 compression function on different numbers of 512 bit blocks when called from the RISC-V software on a Murax SoC and on a Murax SoC with a `SHA256` accelerator (all using the "fixed input length" optimization in software, i.e., cost for SHA-256 padding is not included).

length" and "pre-computation" software optimizations). For the key generation, signing and verification operations, the `SHA256` module accounts for an about 3.8× speedup in the XMSS scheme.

To further accelerate the XMSS computations in an efficient way, in the following we describe the XMSS-specific hardware accelerators that we developed. We first describe an XMSS-specific SHA-256 accelerator, which performs fixed-length SHA-256 padding and provides optional internal storage for one pre-computed state in hardware. Then, we describe how we use this XMSS-specific SHA-256 accelerator as building-block for larger hardware accelerators: An accelerator for WOTS-chain computations and an accelerator for XMSS-leaf generation including WOTS and L-tree computations.

5.2 XMSS-Specific SHA-256 Accelerator

In Section 4, we proposed two software optimizations for the XMSS scheme: "fixed input length" for accelerating SHA-256 computations on 768 bit and 1024 bit inputs and "pre-computation" for acceleration of the function prf(). For hardware acceleration, we introduced a general-purpose SHA-256 hardware module in Section 5.1, which replaces the SHA-256 compression function and thus naturally supports the "fixed input length" optimization and the "pre-computation" optimization of the software implementation. However, both of the optimizations require to repeatedly transfer the same data, i.e., padding or the pre-computed state, to the `SHA256` module, e.g. the "pre-computation" optimization requires to transfer the pre-computed internal state for each prf()

computation. These data transfers introduce an overhead. To eliminate this overhead and as building block for the hardware accelerator modules described in the following sub-sections, we developed an XMSS-specific SHA-256 accelerator, the `SHA256XMSS` module. It has a similar functionality as the general `SHA256` module; however, the `SHA256XMSS` module supports both of the software optimizations internally: It only accepts complete input data blocks of size 768 bit or 1024 bit and adds the SHA-256 padding in hardware. In addition, it provides an optional internal 256 bit register for storing and replaying a pre-computed state.

Implementation. We used the `SHA256` module as building block for the implementation of the `SHA256XMSS` module. All the SHA-256 compression computations in `SHA256XMSS` are done by interacting with the `SHA256` module. In order to handle larger input blocks, the `data_in` port of the `SHA256XMSS` module is 1024 bit wide. The `SHA256XMSS` module has an additional state machine to autonomously perform two or three compression-function iterations (depending on the input length). The state machine also takes care of appending the pre-computed SHA-256 padding to the input data before the last compression function computation. To select between different input lengths, the `SHA256XMSS` module has a `message_length` input signal (low for 768 bit, high for 1024 bit). To support the "pre-computation" optimization, the `SHA256XMSS` module has a similar interface as described for the `SHA256` module in Section 5.1, which allows to set the internal state from software.

To further support the pre-computation functionality in hardware, a 256 bit register can optionally be activated at synthesis time to the `SHA256XMSS` module for storing the fixed internal state. An input signal `store_intermediate` is added for requesting to store the result of the first compression-function iteration in the internal 256 bit register. An input signal `continue_intermediate` is added for requesting to use the previously stored internal state instead of the first compression iteration. The pre-computation functionality can be enabled (marked as "+ `PRECOMP`" in the tables) or disabled at synthesis time in order to save hardware resources for a time-area trade-off.

To reduce the latency of data transfer between the `SHA256XMSS` module and the software, the `SHA256XMSS` module starts computation once the first input data block (512 bit) is received. While the `SHA256XMSS` module is operating on the first input block, the software sends the rest of the input data. An input signal `second_block_available` is added which goes high when the rest of the input data is received. When a valid `second_block_available` signal is received, the `SHA256XMSS` module starts the computation on the rest of the input data once it finishes the previous computation.

When the `SHA256XMSS` module is added to the Murax SoC as a hardware accelerator, it provides a `SHA256` accelerator as well since the `SHA256` module is used as its building block. To achieve this, a hardware wrapper is designed (as shown in Figure 5) which includes both the `SHA256XMSS` module and the `SHA256` module. Apart from the control signals and input data, the bridge module `Apb3Bridge` also takes care of forwarding a 3 bit `cmd` signal from the software to the hardware wrapper. Depending on the value of `cmd`, the hardware

Design	Cycles	Area (ALM)	Reg.	Fmax (MHz)	Time (µs)	Time×Area (relative)	Speedup
			two 512 bit blocks				
SHA256XMSS	128	1680	2070	89.7	1.43	1.77	1.00
+ PRECOMP	64	1900	2320	98.3	0.651	1.00	2.19
			three 512 bit blocks				
SHA256XMSS	192	1680	2070	89.7	2.14	—	—
			hash768				
Murax	10,700	1350	1660	152	70.4	16.0	1.00
+ SHA256XMSS	274	3490	4890	97.8	2.80	1.06	25.1
+ PRECOMP	247	3660	5170	97.6	2.53	1.00	27.8
			hash1024				
Murax	15,600	1350	1660	152	102	13.1	1.00
+ SHA256XMSS	458	3490	4890	97.8	4.68	1.00	21.9

Table 3: Performance of hardware module `SHA256XMSS` and performance comparisons of SHA-256 computations for 768 bit and 1024 bit (functions `hash768` and `hash1024`) when called from the RISC-V software on a Murax SoC and on a Murax SoC with a `SHA256XMSS` accelerator.

wrapper further dispatches the signals to the corresponding hardware module (`SHA256XMSS` or `SHA256`) and triggers the computation. Similarly, based on the `cmd` value, the output data from the corresponding module is returned. The design of the hardware wrapper brings the flexibility that the `SHA256XMSS` module can not only accelerate XMSS-specific SHA-256 function calls, but also general SHA-256 function calls that accept variable length inputs, which may be helpful for some other applications running in the system.

We replaced most of the SHA-256 function calls in the XMSS reference implementation with calls to the `SHA256XMSS` module. The software interface to `SHA256XMSS` is implemented in a function called `sha256xmss`. This function takes a `data_in` pointer to the input data block, a `message_length` flag, a `store_intermediate` flag, and a `continue_intermediate` flag as input and returns the 256 bit result in a `data_out` buffer.

Evaluation. Table 3 shows the performance, resource requirements, and maximum frequency of the `SHA256XMSS` module. When the pre-computation functionality is not enabled, it requires 128 cycles and 192 cycles respectively (one cycle per round) for computing the hash digests for input messages of size 768 bit and 1024 bit. When the pre-computation functionality of the `SHA256XMSS` module is enabled, the cycle count for computing the hash digests for input messages of size 768 bit is halved, because only one 512 bit block needs to be compressed instead of two. However, storing the pre-computed state to achieve this speedup increases ALM and register requirements.

A comparison of the performance and resource requirements of the hash768 and hash1024 function calls for the plain Murax design with the "Murax + SHA256XMSS" design is also shown in Table 3. When the pre-computation functionality of the SHA256XMSS module is enabled, one hash768 call within design "Murax + SHA256XMSS + PRECOMP" obtains a speedup of around 27.8× over the plain Murax design. However, the time-area product only improves by a factor of about 16.0×.

Table 6 shows the performance impact of the SHA256XMSS module on XMSS key generation, signing, and verification (design Murax compared to "Murax + SHA256XMSS" and "Murax + SHA256XMSS + PRECOMP"). For these operations, the SHA256XMSS module accounts for an about 5.4× speedup with pre-computation enabled. Compared to adding a SHA256 module to the Murax SoC, this gives an over 1.4× speedup in accelerating XMSS computations.

5.3 WOTS-chain Accelerator

The SHA256XMSS module provides a significant speedup to the XMSS computations. However, since inputs and outputs need to be written to and read from the SHA256XMSS module frequently, the raw speedup of the SHA-256 accelerator cannot fully be exploited: It actually takes more time to send the inputs to and to read the results from the accelerator than the accelerator requires for the SHA-256 operations. This overhead can significantly be reduced by performing several SHA-256 operations consecutively in hardware. In this case, the hardware accelerator needs to be able to prepare some of the inputs by itself.

The WOTS chain computations are an ideal candidate for such an optimization, because the prf() computations performed in each chain step share a large amount of their inputs (only a few bytes are modified in the address fields for each prf() computation) and the f() computations use previous hash-function outputs. Therefore, we implemented the hardware module Chain as dedicated hardware accelerator for the WOTS chain computations to accelerate the work of the software function gen_chain (see Figure 2).

Implementation. One building block of the Chain module is the Step module, which implements the prf() and the keyed hash-function f() (see Section 2.1) in hardware. The Step module takes in a 256 bit XMSS public seed, a 256 bit data string and a 256 bit address string as input and returns a 256 bit output. Within Step, two prf() computations and one f() computation are carried out in sequence using the hardware modules PRF and F. PRF and F are both implemented by interfacing with a SHA256XMSS module described in Section 5.2. The result generated by the first prf() computation is buffered in a 256 bit register and used later as hash-function key. Similarly the second prf() computation result is buffered in a 256 bit register MASK. The 256 bit input data then gets XOR-ed with MASK and sent to the final f() computation together with the previously computed hash key. The result of the f() computation is returned as the output of the Step module.

The hardware module `Chain` repeatedly uses the `Step` module. It has two input ports `chain_start` and `chain_end`, defining the start and end step for the WOTS chain computation respectively, e.g., 0 and $w - 1$ when used in WOTS key generation. Each step in the `Chain` module uses its step index as its input address and the output from the previous step as its input data. The result from the last step is returned as the result of the `Chain` module.

The "pre-computation" optimization (see Section 4.2) can be optionally enabled for the `SHA256XMSS` module before synthesis. To enable the optimization, the `store_intermediate` port of the `SHA256XMSS` module is set to high for the very first prf() computation to request the `SHA256XMSS` module to store the result of the first compression-function in its internal 256 bit register. For all the following prf() computations, the input port `continue_intermediate` of the `SHA256XMSS` module is raised high to request the usage of the previously stored internal state.

When the `Chain` module is added to the Murax SoC as a hardware accelerator, it provides a `SHA256XMSS` or a `SHA256` accelerator as well since these modules are used as building blocks in `Chain`. A similar hardware wrapper as described for the `SHA256XMSS` accelerator in Section 5.2 is used, which wraps the `Chain` module, the `SHA256XMSS` module, and the `SHA256` module.

We replaced all the WOTS-chain function calls in function `gen_chain` of the XMSS reference implementation (see Figure 2) with calls to the `Chain` module. The software interface is similar to the previously defined interfaces: The function `chain` has as arguments a `data` pointer to the input data string, a `key` pointer to the input key, and an `address` pointer to the address array for the inputs and a `data_out` pointer to the output buffer for the results.

Evaluation. Table 4 shows performance, resource requirements, and maximum frequency of the `Chain` module. Enabling the "pre-computation" optimization ("+ `PRECOMP`") results in a 1.53× speedup for the chain computations in hardware.

A comparison between the pure software and the software/hardware performance of the function `gen_chain` is also provided in Table 4. When `gen_chain` is called in the design "`Murax` + `Chain` + `PRECOMP`", a speedup of around 66.5× is achieved compared to the pure software implementation using the `Murax` design.

Table 6 shows the performance impact of the `Chain` module on XMSS key generation, signing, and verification (`Murax` compared to "`Murax` + `Chain`" and "`Murax` + `Chain` + `PRECOMP`"). Note that since the `Chain` accelerator provides a `SHA256XMSS` accelerator as well, when a `Chain` module is added to the Murax SoC, apart from the function `gen_chain`, the `hash768` and `hash1024` functions are also accelerated. The acceleration of the `Chain` module leads to a 23.9× speedup for both key generation and signing and a 17.5× speedup for verification when the pre-computation functionality is enabled. These speedups achieved are much higher compared to those achieved in the design with a `SHA256XMSS`/`SHA256` accelerator, as shown in Table 6.

Design	Cycles	Area (ALM)	Reg.	Fmax (MHz)	Time (µs)	Time×Area (relative)	Speedup
`Chain`	5960	1940	3060	91.0	65.5	1.30	1.00
+ `PRECOMP`	4100	2170	3320	96.0	42.7	1.00	1.53
`Murax`	530,000	1350	1660	152	3490	31.4	1.00
+ `Chain`	6910	4350	6220	91.6	75.4	1.32	46.2
+ `PRECOMP`	4990	4560	6460	95.2	52.4	1.00	66.5

Table 4: Performance of the hardware module `Chain` and performance comparisons of calling the `gen_chain` function from the RISC-V software on a Murax SoC and on a Murax SoC with a `Chain` accelerator, with parameters $n = 32$ and $w = 16$.

5.4 XMSS-leaf Generation Accelerator

When the `Chain` module is used to compute WOTS chains, the IO requirements are still quite high: For each WOTS key generation, the 256 bit WOTS private key and a 256 bit starting address need to be transferred to the `Chain` module for l times, although their inputs only differ in a few bytes of the address, and l WOTS chain public keys each of 256 bit need to be transferred back.

To reduce this communication overhead, we implemented a WOTS-leaf accelerator module, replacing the software function `treehash` (see Figure 2). The `Leaf` module only requires a 256 bit address (leaf index), a 256 bit secret seed, and a 256 bit XMSS public seed as input. After the `Leaf` module finished computation, the 256 bit L-tree root hash value is returned as the output.

Implementation. As shown in Figure 5, the `Leaf` module is built upon two sub-modules: a `WOTS` module and an `L-tree` module. The `WOTS` module uses the `Chain` module described in Section 5.3 to compute the WOTS chains and returns l 256 bit strings as the WOTS public key. Then, these l values are pairwise hashed together as described in Section 2.1 by the `L-tree` module. Finally, the output of the `L-tree` module (the root of the L-tree) is returned as the output of the `Leaf` module.

The `WOTS` module first computes the secret keys for each WOTS chain using a `PRF_priv` module iteratively for l times. As opposed to the prf() computations during the WOTS chain, L-tree, and Merkle tree computations, the `PRF_priv` module takes a private, not a public seed as input. For each iteration, the corresponding address is computed and sent to the `PRF_priv` module as input as well. When the `PRF_priv` module finishes, its output is written to a dual-port memory `mem`, which has depth l and width 256 bit. Once the secret keys for the l WOTS chains have been computed and written to `mem`, the WOTS public key computation begins. This is done by iteratively using the `Chain` module (see Section 5.3) for l times: First, a read request with the chain index as address is issued to `mem`, then the output of the memory is sent to the input data port of the `Chain` module together with an address (the chain index) and the XMSS public seed. The output of the `Chain` module is written back to `mem`, overwriting the previously stored data.

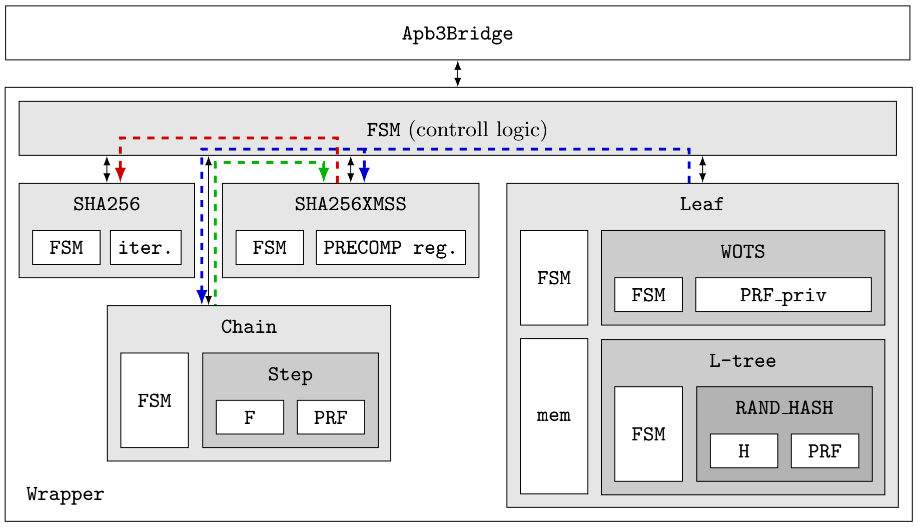

Fig. 5: Diagram of the `Leaf` accelerator wrapper including all the accelerator modules. (control logic is simplified). The `SHA256XMSS` module uses `SHA256`, the `Chain` module uses `SHA256XMSS`, and the `Leaf` module uses `Chain` and `SHA256XMSS`.

Once the WOTS public key computation finishes, the `L-tree` module begins its work. The building block of the `L-tree` module is a `RAND_HASH` module which implements the tree-hash function as described in Section 2.1. It takes in a 256 bit XMSS public seed, two 256 bit data strings, and a 256 bit address string as input and returns a 256 bit output. Within the hardware module `RAND_HASH`, three prf() and one h() computations are carried out in sequence using the modules `PRF` and `H`. The result generated by the first prf() computation is buffered as the 256 bit key while the results from the following prf() computations are buffered as the two 256 bit masks. The two 256 bit input data strings then get each XOR-ed with a mask and sent to the final h() computation together with the previously computed key. The result of the h() computation is returned as the output of the `RAND_HASH` module.

The `L-tree` module constructs the nodes on the first level by first reading out two adjacent leaf nodes from the dual-port memory `mem` by issuing two simultaneous read requests to adjacent memory addresses. The memory outputs are sent to the `RAND_HASH` module as input data. Once `RAND_HASH` finishes computation, the result is written back to `mem` in order (starting from memory address 0). Since the L-tree is not a complete binary hash tree, it occasionally happens that there is a last node on one level that does not have a sibling node. This node is read out from `mem` and immediately written back to the next available memory address. This pattern of computation is repeated until the root of the L-tree is reached. This root is returned as the output of the `Leaf` module.

In order to minimize the resource usage of the `Leaf` module, all the hash computations are done by interfacing with the same `SHA256XMSS` module. Figure 5 shows a diagram of the main building blocks of the `Leaf` module. The "pre-

Design	Cycles	Area (ALM)	Reg.	FMax (MHz)	Time (ms)	Time×Area (relative)	Speedup
`Leaf`	447×10^3	4060	6270	86.1	5.20	1.23	1.00
+ `PRECOMP`	306×10^3	4820	6840	92.8	3.30	1.00	1.58
`Murax`	27.7×10^6	1350	1660	152	182	18.5	1.00
+ `Leaf`	450×10^3	6460	9270	86.6	5.19	1.45	35.0
+ `PRECOMP`	309×10^3	6500	9540	93.1	3.32	1.00	54.8

Table 5: Performance of the hardware module `Leaf` and performance comparisons of calling the `treehash` function from the RISC-V software on a Murax SoC and on a Murax SoC with a `Leaf` accelerator, with parameters $n = 32$ and $w = 16$.

computation" optimization for the prf() computations again can be enabled for the `SHA256XMSS` module before synthesis. When the `Leaf` module is added to the Murax SoC as a hardware accelerator, it also provides a `Chain`, a `SHA256XMSS`, and a `SHA256` accelerator since these modules are all used as building blocks in the `Leaf` module.

The `Leaf` module is called from `treehash` (or the respective BDS functions [6]) instead of functions `wots_pkgen` and `l_tree` in the XMSS reference implementation (see Figure 2). As interface to the `Leaf` module, we provide the software function `leaf`. This function has as arguments a `secret_seed` pointer to the secret key for `PRF_priv`, a `public_seed` pointer to the XMSS public seed, and a `address` pointer to the address array for the inputs and a pointer `data_out` for the result.

Evaluation. Table 5 shows performance, resource requirements, and maximum frequency of the `Leaf` module. Enabling the "pre-computation" optimization ("+ `PRECOMP`") gives a 1.58× speedup at the cost of a small area overhead. Calling the accelerator in function `treehash` in the design "`Murax` + `Leaf` + `PRECOMP`" brings a 54.8× speedup over the pure software implementation on the plain `Murax` design. More importantly, as we can see from the Table (row "`Leaf` + `PRECOMP`" and "`Murax` + `Leaf` + `PRECOMP`"), the IO overhead is no longer impacting the performance of the hardware accelerator `Leaf`.

Table 6 shows the performance impact of the `Leaf` module on XMSS key generation, signing and verification (`Murax` compared with "`Murax` + `Leaf`" and "`Murax` + `Leaf` + `PRECOMP`"). When a `Leaf` module is added in the Murax SoC, it accelerates the functions `treehash`, `gen_chain`, `hash768` and `hash1024` in XMSS. For the key-generation operation, the `Leaf` module accounts for a 54.1× speedup with "`PRECOMP`" enabled. The `Leaf` module is not used during verification and hence does not affect its execution time. The BDS signing algorithm [6] does make use of the `Leaf` accelerator: For signing the first 16 XMSS leaves, on average a 42.8× speedup is achieved.

Design	Cycles	Reg.	Area (ALM)	BRAM (Blocks)	FMax (MHz)	Time	Time× Area	Speedup
			key generation					
`Murax`	28,300,000,000	1660	1350	132	152	186 s	11.2	1.00
+ `SHA256`	4,870,000,000	3880	2860	132	99.9	48.8 s	6.23	3.82
+ `SHA256XMSS`	3,810,000,000	4890	3490	132	97.8	39.0 s	6.09	4.78
+ `PRECOMP`	3,350,000,000	5170	3660	132	97.6	34.3 s	5.60	5.43
+ `Chain`	912,000,000	6220	4350	132	91.6	9.96 s	1.93	18.7
+ `PRECOMP`	742,000,000	6460	4560	132	95.2	7.80 s	1.59	23.9
+ `Leaf`	466,000,000	9270	6460	145	86.6	5.38 s	1.55	34.6
+ `PRECOMP`	320,000,000	9540	6500	145	93.1	3.44 s	1.00	54.1
			signing (average of the first 16 XMSS leaf signatures)					
`Murax`	64,800,000	1660	1350	132	152	426 ms	8.85	1.00
+ `SHA256`	11,200,000	3880	2860	132	99.9	112 ms	4.93	3.81
+ `SHA256XMSS`	8,750,000	4890	3490	132	97.8	89.5 ms	4.83	4.76
+ `PRECOMP`	7,700,000	5170	3660	132	97.6	78.8 ms	4.45	5.40
+ `Chain`	2,070,000	6220	4350	132	91.6	22.6 ms	1.52	18.9
+ `PRECOMP`	1,700,000	6460	4560	132	95.2	17.8 ms	1.26	23.9
+ `Leaf`	1,250,000	9270	6460	145	86.6	14.4 ms	1.44	29.5
+ `PRECOMP`	926,000	9540	6500	145	93.1	9.95 ms	1.00	42.8
			verification					
`Murax`	15,200,000	1660	1350	132	152	99.6 ms	5.17	1.00
+ `SHA256`	2,610,000	3880	2860	132	99.9	26.1 ms	2.88	3.81
+ `SHA256XMSS`	2,060,000	4890	3490	132	97.8	21.1 ms	2.84	4.73
+ `PRECOMP`	1,800,000	5170	3660	132	97.6	18.5 ms	2.61	5.39
+ `Chain`	649,000	6220	4350	132	91.6	7.08 ms	1.19	14.1
+ `PRECOMP`	541,000	6460	4560	132	95.2	5.68 ms	1.00	17.5
+ `Leaf`	649,000	9270	6460	145	86.6	7.49 ms	1.87	13.3
+ `PRECOMP`	541,000	9540	6500	145	93.1	5.80 ms	1.46	17.2

Table 6: Time and resource comparison for key generation, signing and verification on a Cyclone V FPGA (all values rounded to three significant figures with $n = 32$, $w = 16$ and $h = 10$). "Time" is computed as quotient of "Cycles" and "FMax"; "Time×Area" is computed based on "Area" and "Time" relative to the time-area product of the respective most efficient design (gray rows); "Speedup" is computed based on "Time" relative to the respective `Murax` design.

6 Performance Evaluation

We measured a peak stack memory usage of 10.7 kB while the total memory usage is below 110 kB (including the binary code with stdlib and the stack; we do not use a heap).

Table 6 shows performance, resource requirements, and maximum frequency of different designs for the XMSS operations: key generation, signing, and verification. Since the runtime of the BDS signature algorithm [6] varies depending on the leaf index, we report the average timing for the first 16 signature leaves of the XMSS tree.

To accelerate the key generation, signing and verification operations in the XMSS scheme, our hardware accelerators ("`SHA256`", "`SHA256XMSS`", "`Chain`" and "`Leaf`") can be added to the Murax SoC, which leads to good speedups as shown in Table 6. In general, the more computations we delegate to hardware accelerators, the more speedup we can achieve in accelerating XMSS computations. However, at the same time, more overhead is introduced in the hardware resource usage, which is a trade-off users can choose depending on their needs. The best time-area product for the expensive key generation and the signing operations is achieved in design "`Murax` + `Leaf`" with "`PRECOMP`" enabled. For the less expensive verification operation, the "`Murax` + `Chain` + `PRECOMP`" design gives the best time-area product.

The maximum frequency for the designs is heavily impacted by our hardware accelerators (which is accounted for in our speedup and time-area product reports), dropping from 152 MHz down to as low as 86.6 MHz. If a high instruction throughput of the Murax SoC is required for an embedded application that is using our XMSS accelerators, a clock-frequency bridge between the APB and our accelerators might be necessary to enable independent clocks; however, this does not have an impact on the wall-clock speedup of our accelerators.

For a tree height of $h = 10$, i.e., a maximum number of $2^h = 1024$ signatures per key pair, the time for XMSS key generation can be as short as only 3.44 s using our hardware accelerators. Even more signatures per key pair are conceivably possible using multi-tree XMSS, as shown in Table 7 (row "XMSSˆMT^b"). By use of our hardware accelerators, we expect a similar speedup in accelerating XMSSˆMT as we achieved in XMSS. Signing and verification computations are very efficient on our hardware-software co-design for all the SHA-256 parameter sets, i.e., $n = 32, w = 16, h = \{10, 16, 20\}$: For $h = 10$, signing takes only 9.95 ms and verification takes only 5.80 ms. For a bigger tree height, e.g., $h = 20$, signing and verification are only slightly more expensive: Signing takes 11.1 ms and verification takes 6.25 ms. Our experiments show that running XMSS is very much feasible on a resource restricted embedded device such as the Murax SoC with the help of efficient dedicated hardware accelerators.

7 Related Work

We first compare our work with a very recent work [8] which shows a similar software-hardware co-design for XMSS. Then, we summarize all the existing FPGA-based implementations on other hash-based signature schemes. Finally, comparisons with implementations of XMSS on other platforms are provided. Detailed comparison results are shown in Table 7.

7.1 Software-Hardware Co-Design of XMSS

In 2019, Ghosh, Misoczki and Sastry [8] proposed a software-hardware co-design of XMSS based on a 32-bit Intel Quark microcontroller and a Stratix IV FPGA. WOTS computations are offloaded to a WOTS hardware engine which uses a

Design	Parameters (n,h,w)	Hash	Feature	Platform	Freq. MHz	KeyGen. $\times10^9$cyc.	Sign $\times10^6$cyc.	Verify $\times10^6$cyc.
CMSS [25]	32,(10x3),8	SHA-512	HW	Virtex-5	170	1.2	3.7	2.2
SPHINCS [1]	—	ChaCha-12	HW	Kintex-7	525	—	0.80	0.035
XMSS [13]	16,10,16	AES-128	AES	SLE78	33	0.62	3.3	0.56
XMSS^b	32,10,16	SHA-256	SW	Intel i5	3200	5.6	13	3.0
XMSS^o	32,10,16	SHA-256	SW-HW	Murax SoC	93	0.32	0.93	0.54
XMSS^b	32,16,16	SHA-256	SW	Intel i5	3200	360	14	3.1
XMSS [8]	32,16,16	Keccak-400	SW	Quark (Q)	32	—	—	26
XMSS [8]	32,16,16	Keccak-400	SW-HW	Q+Stratix IV	32	—	—	4.8
XMSS^o	32,16,16	SHA-256	SW	Murax SoC	152	1800	70	15
XMSS^o	32,16,16	SHA-256	SW-HW	Murax SoC	93	21	0.99	0.56
XMSS^b	32,20,16	SHA-256	SW	Intel i5	3200	5700	15	3.2
XMSS^o	32,20,16	SHA-256	SW-HW	Murax SoC	93	330	1.0	0.58
XMSSˆMT^b	32,(10x2),16	ChaCha-20	SW	Cortex-M3	32	9.6	18	5.7
XMSSˆMT^b	32,(10x2),16	ChaCha-20	SW	Murax SoC	152	14	28	8.2

Table 7: Comparison with related work. All the tests running on Murax SoC with SW-HW feature is based on the "`Murax` + `Leaf` + `PRECOMP`" design. ^{b} shows our benchmarks and ^{o} means our work.

general-purpose Keccak-400 hash core as building block. In their design, generating one WOTS key pair takes 355,925 cycles, consuming 2963 combinational logic cells and 2337 register cells. This hardware enginee has the same functionality as our `WOTS` module described in Section 5.4. In our design, the `WOTS` module (with "+ `PRECOMP`") takes 279,388 cycles for generating a key pair. The synthesis result of our `WOTS` module on the same FPGA reports a usage of 2397 combinatorial logic cells and 3294 register cells. However, as shown in [8], keccak-400 has a 6× smaller Time×Area compared to SHA-256 when implemented on a 14nm technology. Given such big differences in the building hash core, a fair comparison between the two WOTS designs is not possible.

By use of the WOTS hardware engine, running the XMSS reference implementation on their software-hardware co-design with $n = 32, h = 16, w = 16$ takes 4.8×10^6 cycles in verification on average (key generation and complete signature generation are not included in their tests), leading to a 5.3× speedup compared to running the design purely on the Quark microcontroller. To achieve a better comparison, we run a full XMSS test with the same parameter set on the "`Murax` + `Leaf` + `PRECOMP`" design. As shown in Table 7, in terms of cycle count, our design achieves a 5× bigger speedup compared to [8] in accelerating the verification operation in XMSS. However, a fair comparison between our work and [8] is not feasible due to the differences in the platforms, the hardware accelerators, the building hash cores, etc.

7.2 Hash-Based Signature Schemes on FPGA

There are currently only a few publications focusing on FPGA hardware implementations of hash-based signature schemes:

In 2011, Shoufan, Huber and Molter presented a cryptoprocessor architecture for the chained Merkle signature scheme (CMSS) [25], which is a successor of the classic Merkle signature scheme (MSS). All the operations, i.e., key generation, signing, and verification are implemented on an FPGA platform. By use of these coprocessors, for parameters $w = 8$, tree height on a CMSS level $h = 10$ and total CMSS levels $T = 3$, the authors report timings of 6.9 s for key generation, 21.5 ms for signing and 13.2 ms for verification. In their design, twelve SHA-512 modules in total are used to parallelize the design for better speedups.

Their implementation, however, is no longer state-of-the-art: They provide none of the additional security features that have been developed for modern hash-based signature schemes like XMSS, LMS [17], and the SPHINCS family [4]. The straightforward hash-based operations are all replaced with more complex operations involving masks and keys computed by pseudorandom functions. Therefore, direct comparisons between the hardware modules among MSS and XMSS cannot be fairly done.

For modern hash-based signature schemes, in 2018, an implementation of the stateless hash-based signature scheme SPHINCS-256 [4] was proposed in [1]. This signature scheme is closely related to XMSS and is a predecessor of the SPHINCS+ signature scheme[13], which is one of the submissions in NIST's PQC standardization process[14]. SPHINCS-256 requires the cryptographic primitives BLAKE-256, BLAKE-512, and ChaCha-12. The authors provide efficient hardware implementations for these primitives and control logic to enable signing, key generation, and signature verification. They report timings of 1.53 ms for signing and 65 µs for verification, but no timings for key generation.

The source code of all these works [1,25] is not freely available. The detailed performance data for the main hardware modules is not provided in the paper either. Lack of access to the source code and detailed performance results make comparisons unfruitful.

7.3 XMSS on Other Platforms

We first benchmarked the original XMSS software implementation (linked against the OpenSSL library) for all the SHA-256 parameter sets on an Intel i5-4570 CPU. The performance results in Table 7 show that running the optimized XMSS software implementation on our software-hardware co-design leads to an over 15× speedup in terms of clock cycles compared to running the implementation on an off-the-shelf Intel i5 CPU. In 2012, Hülsing, Busold, and Buchmann presented an XMSS-based implementation [13] on a 16-bit Infineon SLE78 microcontroller, including key generation, signing and verification. The hash functions are implemented by use of the embedded AES-128 co-processor. Performance results for XMSS with $n = 16, h = 10$ and $w = 16$ maintaining a classical security level of 78 bit is provided. However, a fair comparison between

[13] `https://sphincs.org/`

[14] `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`

our work and [13] is not feasible since the security parameters used in [13] are already outdated.

The practicability of running SPHINCS [4] on a 32-bit ARM Cortex-M3 processor is demonstrated in [15]. For comparison, they also implemented the multi-tree version of XMSS(XMSSˆMT) on the same platform. Chacha-20 is used as the building hash function in their design. To get a fair comparison between the performance of the Murax SoC and a Cortex-M3 processor, we compiled a pure C-version of the code from [4] for both an ARM Cortex-M3 processor and the Murax SoC and then measured the performance of XMSSˆMT on these two platforms. As shown in Table 7, running the same test on the Murax SoC gives a less than 50% slowdown in terms of cycle count compared to an off-the-shelf ARM Cortex-M3 processor while the Murax SoC can run at an about 5× higher clock frequency. This shows that the performance of the Murax SoC is comparable to the Cortex-M3. Moreover, this test shows the feasibility of running the XMSSˆMT with a bigger $h = 20$ on the Murax SoC.

8 Conclusion

In this paper, we presented the first software-hardware co-design for the XMSS scheme on a RISC-V-based embedded system. We first proposed two software optimizations targeting the SHA-256 function for the XMSS reference software implementation, and then developed several hardware accelerators to speed up the most expensive operations in XMSS, including a general-purpose SHA-256 accelerator, an XMSS-specific SHA-256 accelerator, a WOTS-chain accelerator and a WOTS-leaf accelerator. The integration of these hardware accelerators to the RISC-V processor brings a significant speedup in running XMSS on our software-hardware co-design compared to the pure software version. Our work shows that embedded devices can remain future-proof by using algorithms such as XMSS to ensure their security, even in the light of practical quantum computers.

Acknowledgments. This work was supported in part by NSF grant 1716541. Part of the research was performed when the second author was affiliated at Fraunhofer Singapore.

References

1. Amiet, D., Curiger, A., Zbinden, P.: FPGA-based accelerator for post-quantum signature scheme SPHINCS-256. IACR Transactions on Cryptographic Hardware and Embedded Systems – CHES 2018 **2018**(1), 18–39 (Feb 2018), Open Access
2. Aysu, A., Schaumont, P.: Precomputation methods for faster and greener post-quantum cryptography on emerging embedded platforms. IACR Cryptology ePrint Archive **2015**, 288 (2015)
3. Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.): Post-Quantum Cryptography. Springer, Heidelberg (2009)

4. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. LNCS, vol. 9056, pp. 368–397. Springer (2015)
5. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS – a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.Y. (ed.) Post-Quantum Cryptography – PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer (2011), second Version, IACR ePrint Report 2011/484
6. Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. In: Buchmann, J., Ding, J. (eds.) Post-Quantum Cryptography – PQCrypto 2008. LNCS, vol. 5299, pp. 63–78. Springer (2008)
7. García, R., Algredo-Badillo, I., Morales-Sandoval, M., Feregrino-Uribe, C., Cumplido, R.: A compact FPGA-based processor for the secure hash algorithm SHA-256. Computers & Electrical Engineering **40**(1), 194–202 (2014)
8. Ghosh, S., Misoczki, R., Sastry, M.R.: Lightweight post-quantum-secure digital signature approach for IoT motes. IACR Cryptology ePrint Archive (2019)
9. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Symposium on the Theory of Computing – STOC 1996. pp. 212–219. ACM (1996)
10. Higginbotham, S.: The rise of RISC - [opinion]. IEEE Spectrum **55**(8), 18 (Aug 2018)
11. Homsirikamol, E., Rogawski, M., Gaj, K.: Throughput vs. area trade-offs in high-speed architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2011. LNCS, vol. 6917, pp. 491–506. Springer (2011)
12. Hülsing, A.: W-OTS+ – shorter signatures for hash-based signature schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) Progress in Cryptology – AFRICACRYPT 2013. LNCS, vol. 7918, pp. 173–188. Springer (2013)
13. Hülsing, A., Busold, C., Buchmann, J.: Forward secure signatures on smart cards. In: International Conference on Selected Areas in Cryptography – SAC 2012. pp. 66–80. Springer (2012)
14. Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. RFC **8391**, 1–74 (2018)
15. Hülsing, A., Rijneveld, J., Schwabe, P.: ARMed SPHINCS. In: Public-Key Cryptography – PKC 2016, pp. 446–470. Springer (2016)
16. Kahri, F., Mestiri, H., Bouallegue, B., Machhout, M.: Efficient FPGA hardware implementation of secure hash function SHA-256/Blake-256. In: Systems, Signals & Devices (SSD), 2015 12th International Multi-Conference on. pp. 1–5. IEEE (2015)
17. McGrew, D., Curcio, M., Fluhrer, S.: Hash-based signatures. cfrg **draft-mcgrew-hash-sigs-1**, 1–60 (2018)
18. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology – CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer (1990)
19. Merritt, R.: Microsoft and Google planning silicon-level security. EE Times Asia (Aug 2018), URL: `https://www.eetasia.com/news/article/18082202-microsoft-and-google-planning-silicon-level-security`
20. NIST: FIPS PUB 180-4: Secure Hash Standard. National Institute of Standards and Technology (2012)
21. NIST: FIPS PUB 186-4: Digital Signature Standard. National Institute of Standards and Technology (2013)

22. Padhi, M., Chaudhari, R.: An optimized pipelined architecture of SHA-256 hash function. In: Embedded Computing and System Design (ISED), 2017 7th International Symposium on. pp. 1–4. IEEE (2017)
23. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Foundations of Computer Science – FOCS '94. pp. 124–134. IEEE (1994)
24. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review **41**(2), 303–332 (1999)
25. Shoufan, A., Huber, N., Molter, H.G.: A novel cryptoprocessor architecture for chained Merkle signature scheme. Microprocessors and Microsystems **35**(1), 34–47 (2011)
26. Teich, J.: Hardware/software codesign: The past, the present, and predicting the future. Proceedings of the IEEE **100**(Special Centennial Issue), 1411–1430 (2012)