No RISC, no Fun: Comparison of Hardware Accelerated Hash Functions for XMSS

Ingo Braun[1], Fabio Campos[1], Steffen Reith[1], and Marc Stöttinger[2]

[1] Department of Computer Science, University of Applied Sciences Wiesbaden, Germany
ingo.braun@student.hs-rm.de
campos@sopmac.de
steffen.reith@hs-rm.de
[2] Continental AG, Germany
marc.stoettinger@continental-corporation.com

Abstract. We investigate multiple implementations of a hash-based digital signature scheme in software and hardware for a RISC-V processor. For this, different instantiations of XMSS by leveraging SHA-256 and SHA-3 are considered. Moreover, we propose various optimisations for accelerating the signature scheme on resource-constrained FPGAs. Compared to the pure software version, the implemented hardware accelerators for SHA-256 and SHA-3 achieve a significant speedup of 25× and 87× respectively for generating 2^{10} key pairs. Signing and verifying with such key pairs achieves a speedup of 17× and 10× in the case of SHA-256, and respectively 55× and 20× for SHA-3. Recently, Wang et al. presented an XMSS-specific software-hardware co-design, resulting in significant speedups. Our general-purpose hardware accelerator for SHA-256 further reduces the calculation cost for signing by 26%, and by 28% for verifying in comparison to results of Wang et al., and achieves a better time-area product for signing (3.3×) and verifying (2.5×).

Keywords: XMSS, RISC-V, hash-based signatures, post-quantum cryptography, FPGA, resource-constrained systems

1 Introduction

Digital signatures are an essential primitive of modern cryptography, providing authenticity, integrity, and non-repudiation. A digital-signature scheme is a cryptographic primitive which allows a signer to generate a key-pair (public and secret keys), sign data using the secret key, and enables a verifier to verify signed data using the public key. Hence, digital signatures have a wide application where the interacting parties might not need to be considered in a environment with relaxed assumptions on trustworthiness. Typical applications are for instance credential-passed access control, secure software download, authentic communication, etc. Typical signature schemes

* Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf

This work was partially supported by Elektrobit Automotive, Erlangen, Germany and the Security & Privacy Competence Center, Frankfurt, Germany.

widely used in practice today are RSA [25], DSA [10], ECDSA [16], and EdDSA [3]. These algorithms can be successfully attacked by a quantum computers since their security relies on the hardness of number-theoretical problems such as factorisation or the discrete-logarithm problem in certain groups [26]. Quantum-safe signature algorithms are required for application domains in which a long-term security for multiple years is required. Promising candidates for the construction of post-quantum signature schemes are hash-based signatures. The reason for this observation is that the best known algorithms for computing hash collisions on classical computers are based on the birthday paradox and give a square-root speedup over a brute-force search. Grover's algorithm [12] gives at most a cube-root speedup as shown in [5], nonetheless according to [2] the cost model applied in [5] does not reflect real-world attack cost.

The idea of hash-based signatures was first proposed by Lamport [19]. Merkle proposed in 1989 the Merkle Signature Scheme (MSS) [22]. This approach starts with a one-time signature scheme and constructs a many-time signature scheme using binary hash trees. One of the biggest disadvantages of hash-based signature schemes is the relatively long execution time of the key generation process. In addition, signature generation and verification is slow compared to currently used signature schemes. The performance of hash-based signature schemes strongly depends on the performance of the used hash function. Therefore, optimizing the underlying hash function is the most promising approach for optimisation efforts. The eXtended Merkle Signature Scheme (XMSS) first proposed in 2011 [8] describes an efficient forward-secure post-quantum stateful signature scheme with minimal security assumptions. It is based on Merkle's approach and on the Generalised Merkle Signature Scheme (GMSS) [6]. SPHINCS+ [4], a stateless hash-based signature framework, and NIST-PQC[1] candidate, which is not the focus of this work, uses many components from XMSS, hence our results might be valuable for future work in this area. We chose XMSS as a study example for our analyses on speeding up hash-based signatures, because it has already been documented in a standardisation manner by the IETF[2].

The RISC-V instruction set architecture (ISA) is free and open for use in all types of implementations without restriction. It is based on reduced-instruction-set computer (RISC) principles. The RISC-V architecture is increasingly used not only in the academic environment but also in an industrial context. Due to its open and free architecture, RISC-V is highly promising for security applications. Especially in the security and trusted computing domain, the use of a transparent processor architecture can reduce the dependencies on chip manufacturers. Since according to [30] security spawns across all sectors (hardware, software, protocols, systems, services, and infrastructures) the only way to improve the general level of protection is deemed to retain or rather to regain control of the entire supply chain.

Related Work. Some aspects of hardware implementations and accelerations of hash-based schemes have been studied in the literature. Shoufan et al. [27] implemented a hardware accelerator for generating Merkle hash trees. Hülsing et al. [15] demonstrated that only a small amount of memory is required to generate and verify a SPHINCS-256

[1] `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`
[2] `https://tools.ietf.org/html/rfc8391`

signature on an ARM Cortex M3. Van der Laan et. al [18] presented an implementation of XMSS on the current Java Card platform, and make suggestions for some improvements. Amiet et al. [1] presented the first FPGA-based hardware accelerator for SPHINCS-256. Ghosh et al. [11] proposed a latency-area optimised hardware-software architecture for XMSS with 128-bit post-quantum security. Wang et al. [28] achieved a significant speedup with a hardware-software co-design implementation for XMSS, where they developed XMSS-specific hardware accelerators.

Our Contribution. In this work, we analyzed software and hardware implementations of SHA-256, and SHA-3 on an FPGA-based RISC-V processor. The implemented hardware accelerators are attached to a RISC-V processor by an instruction-set extension approach. In addition, we discuss various architectural approaches and their impact on the performance. We provide a comparative performance analysis based on clock cycles of the instantiated versions of XMSS. Furthermore, we show that a generic optimisation of the underlying hash function in XMSS leads to significant speedup. Our implementation is publicly available at `https://doi.org/10.5281/zenodo.3556239`. Due to the selected approach, the resulting software-hardware co-design can be integrated into other hardware architectures with negligible additional effort.

Organisation. The remainder of this paper is organised as follows. We start with an introduction to Merkle Trees, Winternitz One-time Signatures, and XMSS in Section 2. In Section 3 we discuss different design criteria and briefly present the implemented hash functions. Section 4 contains a description of our software and hardware implementation. Thereafter, we summarise our experimental results in Section 5 and give a conclusion in Section 6.

2 Hash-based Signature Schemes

Hash-based signature schemes, whose security is only based on properties of the underlying cryptographic hash functions, are among the most promising candidates for quantum-safe digital signature schemes. While earlier hash-based schemes did not meet practical time and space requirements, current schemes like LMS [20], XMSS, and SPHINCS+ are faster and provide reasonably small signatures. In this section we introduce the core elements and components of XMSS used in our case study.

Lamport et al. [19] proposed the most basic one-time hash-based signature scheme (LOTS). Given a security parameter n and a one-way function $f : \{0,1\}^n \rightarrow \{0,1\}^n$, the following one-time scheme allows to sign a message M of m bits. The secret key consists of $2m$ random values $x_b^i \in \{0,1\}^n$ and the corresponding public key $y_b^i = f(x_b^i)$, where $b \in \{0,1\}$ and $1 \leq i \leq m$. The signature σ for a message $M = M_1 ... M_m$ is then $\sigma = (x_{M_1}^1, ..., x_{M_m}^m)$, which can be verified by checking $y_{M_i}^i = f(x_{M_i}^i)$ for $1 \leq i \leq m$. To sign messages of arbitrary length, they further proposed to use the *hash-then-sign* construction.

2.1 Winternitz One-time Signature Scheme

The main idea of the Winternitz one-time signature scheme (WOTS) [22] is to sign multiple bits by calculating the public key based on a function chain. Let n be the security parameter, w the "Winternitz parameter", and $f : \{0,1\}^* \to \{0,1\}^n$ a one-way function, then the key generation is processed as shown in Algorithm 1. Thereby f^{w-1} should be interpreted as the $(w-1)^{\text{th}}$ iteration of the one-way function f.

Algorithm 1: Key generation.

Input : security parameter n, Winternitz parameter w.
Output: one-time key pair: (secret key X, public key Y).

$\ell_1 \leftarrow \lceil n/\log_2(w) \rceil$
$\ell_2 \leftarrow \lfloor \log_2(\ell_1(w-1))/\log_2(w) \rfloor + 1$
$\ell \leftarrow \ell_1 + \ell_2$
for $i = 0, ..., \ell - 1$ **do**
  $x_i \xleftarrow{\$} \{0,1\}^n$ // sampled uniformly at random
  $y_i \leftarrow f^{w-1}(x_i)$ // the $(w-1)^{\text{th}}$ iteration of f
return $((x_0, x_1, ..., x_{\ell-1}), (y_0, y_1, ..., y_{\ell-1}))$

To avoid some trivial attacks a checksum C is calculated, if necessary padded with zeros such that the length n is divisible by $\log_2(w)$, and appended to the message, as shown in Algorithm 2 in line 5-7. The signature is constructed by mapping the chunks of M' to intermediary values of the respective function chain.

Algorithm 2: Signing.

Input : message M, secret key X, security parameter n, Winternitz parameter w.
Output: signature σ.

1 $\ell_1 \leftarrow \lceil n/\log_2(w) \rceil$
2 $\ell_2 \leftarrow \lfloor \log_2(\ell_1(w-1))/\log_2(w) \rfloor + 1$
3 $\ell \leftarrow \ell_1 + \ell_2$
4 $(M_0, M_1, ..., M_{\ell_1}) \leftarrow \mathbf{split}(M)$ // split M into $\log_2(w)$-bit chunks
5 $C \leftarrow \sum_{i=0}^{\ell_1 - 1} w - 1 - M_i$
6 $C \leftarrow \mathbf{pad}(C)$ // pad C with zeros if necessary
7 $M' \leftarrow M \,||\, C$ // concatenate M and C
8 $(M'_0, M'_1, ..., M'_\ell) \leftarrow \mathbf{split}(M')$ // split M' into $\log_2 w$-bit chunks
9 **for** $i = 0, ..., \ell - 1$ **do**
10 $\sigma_i \leftarrow f^{M'_i}(x_i)$
11 **return** $(\sigma_0, \sigma_1, ..., \sigma_{\ell-1})$

Unlike in LOTS, in WOTS the public key can be calculated directly from the signature, as shown in Algorithm 3.

Assuming f is a one-way, collision-resistant and preimage resistant function, this scheme is existentially unforgeable under chosen message attacks (EU-CMA) as shown in [9]. In [13] Hülsing et al. proposed WOTS$^+$, a slight modification of the chaining function by adding a random bitmask r_i for each iteration, such that $f^0(x) = x$, and $f^i(x) = f(f^{i-1}(x) \oplus r_i)$. This modification uses tweakable hash functions [4] and eliminates the requirement for a collision resistant hash function.

Algorithm 3: Verifying.

Input : signature σ, message M, public key Y, security parameter n, Winternitz paramater w.
Output: valid or invalid.

$\ell_1 \leftarrow \lceil n/\log_2(w) \rceil$
$\ell_2 \leftarrow \lfloor \log_2(\ell_1(w-1))/\log_2(w) \rfloor + 1$
$\ell \leftarrow \ell_1 + \ell_2$
$(M_0, M_1, ..., M_{\ell_1}) \leftarrow \textbf{split}(M)$ // split M into $\log_2(w)$-bit chunks
$C \leftarrow \sum_{i=0}^{\ell_1 - 1} w - 1 - M_i$
$C \leftarrow \textbf{pad}(C)$ // pad C with zeros if necessary
$M' \leftarrow M \,||\, C$ // concatenate M and C
$(M'_0, M'_1, ..., M'_\ell) \leftarrow \textbf{split}(M')$ // split M' into $\log_2 w$-bits chunks
for $i = 0, ..., \ell - 1$ **do**
    if $((f^{w-1-M'_i}(\sigma_i)) \neq (f^{w-1}(y_i)))$ **then**
        return invalid
return valid

2.2 Merkle Trees

Based on the idea of one-time signature schemes, Merkle's approach [22] was to construct a balanced binary tree (a so-called *Merkle tree*) using a given hash function to enable the use of a single public key (root of the tree) for verifying several messages. Therefore, a signer generates 2^H one-time key pairs (X_j, Y_j) where $0 \leq j < 2^H$ for a selected $H \in \mathbb{N}$ and $H \geq 2$. The leaves of the tree are the digests $f(Y_j)$ for $0 \leq j < 2^H$. H defines the height of the resulting binary tree whose inner nodes are represented by the values computed as $n = f(n_l \,||\, n_r)$, where n_l and n_r are the values of the left and right child of n. To verify a leaf at index i, one simply needs $f(Y_i)$, i, and the authentication path of i, which is a sequence of H nodes. This authentication path contains the siblings of all the nodes on the path between leaf i and the root, and thus enables the verifier to recompute and validate the root/public key.

2.3 XMSS

XMSS was introduced in 2011 [8] based on the concept of Merkle's tree construction [21] and on the Generalised Merkle Signature Scheme (GMSS) [6]. XMSS describes an efficient post-quantum stateful signature scheme with minimal security assumptions. One of the main advantages of XMSS with WOTS$^+$ is that it does not require a collision-resistant one-way function.

XMSS introduces a hash function address scheme to uniquely identify each individual calculation step. The calculation of the inner nodes of an XMSS tree uses a tweakable hash function and works as follows. First, an n-byte key K and two n-byte masks are computed using a keyed pseudorandom function $\text{PRF}_K : \{0,1\}^{256} \rightarrow \{0,1\}^{256}$. The two lower nodes in a XMSS tree are then XOR-ed to the corresponding mask, concatenated, and used as input for a keyed hash function $\text{h}_K : \{0,1\}^{512} \rightarrow \{0,1\}^{256}$. So in general, the authenticity of many WOTS$^+$ public keys is reduced to one XMSS public key

using a binary Merkle tree. The i-th XMSS signature $\sigma_i = (i, \sigma_{WOTS^+}, (a_0, ..., a_{H-1}))$ includes an index i of the current leaf, the one-time WOTS$^+$ signature σ_{WOTS^+}, and the authentication sequence $(a_0, ..., a_{H-1})$, which contains one node in each layer of the hash tree.

Using a pseudo-random function (PRF) as proposed in [7], the data space for storing the private key can be reduced by storing only the seed of a PRF. Let $\text{PRF} : \{0,1\}^n \rightarrow \{0,1\}^n \times \{0,1\}^n$ be a secure pseudo random number generator which takes an n-bit S_{in} as input and outputs a n-bit random number R and a n-bit seed. Assuming that the calculation of a previous seed from the current seed ($S_{out} \xrightarrow{?} S_{in}$) is unfeasible, therefore using such a PRF leads to the additional benefit of forward security.

XMSS also proposes a multi-tree version (XMSS^{MT}) where the leaf nodes of a higher-level tree are used to sign the roots of lower trees (see [14] for more details).

Parameter Sets. In order to achieve certain post-quantum security levels, RFC 8391 [14] defines parameter sets for the hash functions in order to provide 128 bit and 256 bit security respectively. We will focus on suitable second-preimage-resistant hash functions with digest lengths of 256 bits, achieving 128 bits of post-quantum security against (second)-preimage attacks. In particular, based on the current reference implementation of XMSS[3] we are interested in two types of functions:

$$\mathcal{H} : \{0,1\}^{4n} \rightarrow \{0,1\}^n \text{ and } \mathcal{F} : \{0,1\}^{3n} \rightarrow \{0,1\}^n$$

Let $n = 256$, thus $\mathcal{H} : \{0,1\}^{1024} \rightarrow \{0,1\}^{256}$, and $\mathcal{F} : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$. Table 1 summarises the required calls to these functions for key generation, signing and verification.

Table 1: Number of calls to $\mathcal{H}$ and $\mathcal{F}$ for $n = 256$, $w = 16$, and $H = 10$.

	KeyGen	Sign	Verify
calls of $\mathcal{H}$	68607	66	76
calls of $\mathcal{F}$	3362813	4970	1623

3 Architectural Aspects

In hash-based signature schemes the sole cryptographic operation is a hash function. The remaining computing time is spent on performing XOR-operations, conversions, and managing the data flow. Thus, from an architectural aspect the data transfer to and from the hash function is very important and will impact the overall performance and latency of the hash-based signature operations. Hence, this is the first option to increase the speed up. The second option is to reduce the latency and increase the throughput of the implementation of the hash-function. In general roughly three paradigms exist on how to integrate a hardware accelerator for cryptographic operations.

[3] `https://github.com/joostrijneveld/xmss-reference`, commit fb7e3f8

The first option is an external independent processing unit outside of the housing of the main processing unit of the system. This option highly depends on the security requirements of the use case. For signature operations this might by applicable for verification or remote attestation services, however in most cases the verification information cannot be trusted, because it is communicated via an insecure channel across the printed circuit board. Hence, it is not a favored option to support a general solution to speed up hash-based signature schemes.

The second option is an integrated co-processor running the cryptographic operation, independent from the main processor. Thereby, concurrency is exploitable to speed up processing a lot of data to achieve a high throughput. However, at the same also the integration is time limiting the throughput, because the data handling between the processors needs to be coordinated. Thus in general, a pipelining approach for the co-processor is applied in this scenario in order to hide the latency of the data transfer between the co-processor and the accelerator. Therefore, in our scenario the hash function would require to have large inputs to exploit the pipelining efficiently, but for a Merkle-Tree based hash signature scheme this is usually not the case.

The third option is to integrate the accelerator into the general-purpose processor via an extension of the instruction set architecture (ISA). This will reduce the latency due to memory access, but is not as efficient as the pipelined co-processor approach in case of processing large amounts of independent data, as the general-purpose processor is not running concurrently to the accelerator. Thus, this approach fits more the scenario that is present with a Merkle-Tree based hash signature scheme. The downside of the ISA approach is that the instruction set is fixed and cannot be changed, if another accelerator is integrated, changes to the instruction set might be needed or an complicated adaption is required. Thus, also the compiler for the processing unit needs to be updated which will affect the development environment. This impact can be minimised by designing a generic API for the ISA that makes at least the class of cryptographic functions easier to exchange. In our case, we are aiming for a general API of the ISA for hash functions. The major speed up with this approach is the reduced latency due to fast data access on the memory.

For further investigation on the performance of various hash algorithms, we chose an integration of the hash algorithm via an instruction-set extension. As discussed before, the integration of a cryptographic accelerator via an ISA with a generic API is in our opinion a good compromise for the trade off between flexibility and performance. In addition, comparison between a software implementation and a hardware implementation is more fair when almost the same memory transfer overhead is required. In the following, we will discuss the different implementation options of the hash algorithms.

4 Implementation

Currently, several open source-instruction set architectures (ISAs) (RISC-V, Open RISC, etc.) are available for designing a RISC processor, on ASIC or FPGA. The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries and is divided into several basic instruction sets and extensions. The integer set is available in three different configurations with 32, 64 and 128-bit width registers,

respectively called *RV32I*, *RV64I*, and *RV128I*. Additionally, the *RV32E* configuration, which is essentially a lightweight *RV32I* with a reduced number of registers is available for small embedded devices. The extensions expand the basic commands with additional commands, such as multiplication and division. A comprehensive description of the instruction set is available in the RISC-V specification [29].

VexRiscv[4] is an open-source 32-bit CPU implementing the RISC-V instruction set. The VexRiscv framework has a modular structure, which can easily be extended by integrating plugins. The VexRiscv used in this work supports the basic command set with the multiplication extension for 32-bit words. It further has a classical five-stage pipeline. The provided ability of skipping pipeline stages enables some speedup in certain cases. Currently two types of System on Chip (SoC) of VexRiscv are available: Briey, which implements almost all features of the VexRiscv processor, and Murax, which only implements the most basic features. In this work we implemented the Murax SoC with the following extensions: multiplication and division extension, a simple branch prediction, and the feature to skip pipeline steps. The simple shifter was replaced by one full featured barrel shifter. For development and test purposes the communication with the processor was realised via UART and JTAG interface for debugging. We developed the hardware accelerators in the high-level hardware description language SpinalHDL. SpinalHDL is based on Scala and translates a hardware description into VHDL/Verilog code.

Moreover a hybrid approach to control the hash-function accelerators was implemented. While start (*hashStart()*) and reset (*hashReset()*) functions are respectively provided by custom instructions, data IO is done directly over the simple memory bus of the Murax SoC (*load–store architecture*) as shown in Figure 1. Thus, input and output are handled by a fixed memory address. In order to take full advantage of the data bus, the memory blocks are divided into 32-bit words. Hence, in the case of SHA-256, the input of 512-bits length is distributed over 16 memory words. In the case of SHA-3, the input of 1088-bits length is distributed over 34 words. The resulting digest is provided by a fixed memory address and consists of 8 memory words of 32 bits. For the purpose of modularity, the provided custom instructions for the currently chosen hash functions are identical. More precisely, the provided custom instruction for starting the compression function (SHA-256) and the permutation function (SHA-3) are the same. Basically, the simultaneous use of both hardware accelerators is possible with minor changes and requires the sum of the respective areas.

We provide two methods for flow control. The first implemented approach stalls the pipeline in order to avoid data hazards. Although this method causes some delay in the entire pipeline, it has a significant advantage in terms of portability compared to the second approach. The second method is based on interrupts and therefore requires the implementation of a corresponding interrupt service routine. The proposed software-hardware co-design focuses on a general architecture which differs only slightly depending on the implemented hash function. In order to minimise space and improve portability without significant loss in perfomance, we only implemented the compression respectively permutation function of the chosen hash functions in hardware. As shown in Figure 2, the pre-computation steps of padding and message scheduling were

[4] `https://github.com/SpinalHDL/VexRiscv`, commit fe385da

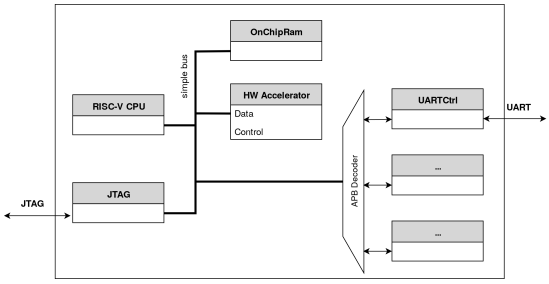

Fig. 1: Schematic Murax SoC with instruction set enabled hardware accelerator.

implemented in software. In order to use many of the outputs simultaneously, the internal registers (message and state) of the hash engine were implemented using flip-flops. Additionally the implemented primitives can operate in parallel, providing high throughput whenever needed. The correctness of the implementation of the algorithms is tested against the available Known Answer Tests (KAT) [23] [24]. We also verified the signing method for deterministic behavior.

Pre-Computation. To improve the performance without the need for additional space, we implemented the pre-computation technique from [28]. Within XMSS, for a given key pair, the first 512-bit block (256-bit domain separator and 256-bit hash-function key) of the input to the pseudo-random function (of type $\mathcal{F} : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$, where $n = 256$) is the same for all calls. Considering this fact, we store the digest of the first 512 bit block at the first call to the pseudorandom function (PRF) and skip this effort by reusing this result in all further calls. The number of calls to the PRF in XMSS is about 70% of the total calls of $\mathcal{F}$ (see Table 1).

This approach can generally be applied if the internal block size of the implemented hash function is less than or equal to 512 bits. Depending on the internal block size of the used hash function, the number of calls to the internal compression respectively permutation function to be saved ($\mathbf{Speedup}_{PRF}$) can be calculated as follows. Let B_{bits} be the internal block size in bits and $\#\text{call}_{PRF}$ be the number of calls to the PRF, then $\mathbf{Speedup}_{PRF} = \lfloor 512 \text{ bits} / B_{bits} \rfloor * \#\text{call}_{PRF}$, where B_{bits} is longer than 512 bits.

In the case of SHA-256, where the 512-bit block fits into a 512 bit SHA-256 internal block, this approach reduces the number of calls to the compression function by half. Although this method can basically be applied in a sponge construction, it cannot be used for SHA-3 due to an internal block size longer than 512 bits.

5 Evaluation

In this section we present the results of our work. We outline the most important performance aspects here, and give a final conclusion in Section 6.

We measured the performance of our software and hardware implementations on a RISC-V implementation (VexRiscv) on a Digilent Nexys 4 DDR board (ARTIX

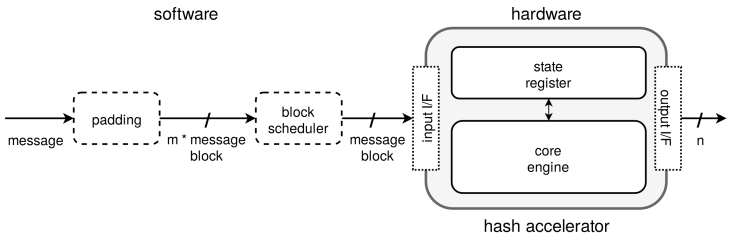

Fig. 2: Schematic representation of the implemented hash functions.

XC7A100 T), using the XMSS[5] reference implementation. Our measurements consider straightforward hardware implementations of SHA-256 and SHA-3. We use the C implementation from Intel's TinyCrypt Cryptographic Library[6] for the software version of SHA-256 and the SUPERCOP[7] for SHA-3[8], both compiled using the GNU Embedded Toolchain[9] with `riscv32-unknown-elf-gcc-9.2.0` using the flags `-march=rv32im -mabi=ilp32 -O3`. The area and the timing report has been obtained after executing place and route in the implementation step on Xilinx Vivado v2017.3 (64-bit) tool. In particular, we performed 100 runs to measure the speed of generating keys, signing messages and verifying signatures. The implemented SHA-3 accelerator needs 8401 slices, and thus 45% more space than the SHA-256 accelerator (5759 slices) and achieves almost the same maximal frequency, as shown in Table 2. The area of both accelerators include 3492 slices for the implemented VexRiscv CPU. The hardware-accelerated hash function requires 1528 clock cycles for calculating $\mathcal{F} : \{0,1\}^{768} \rightarrow \{0,1\}^{256}$ in the case of SHA-256 and 746 clock cycles in the case of SHA-3. In the applied version of SHA-3 the internal block (rate) is 1088 bits of length and therefore it only requires one call to the permutation function. Whereas SHA-256 requires two calls to the compression function, due to the internal block size of 512 bits.

A comparison of the results of our hardware and software implementation of SHA-256 and SHA-3 is shown in Table 3. Ghosh et al. [11] presented a lightweight solution based on XMSS using a general-purpose Keccak-400 hash function. Due to the smaller internal state of Keccak-400 and its lower security level, a comparison of the results is only partially possible, as presented in Table 2. Wang et al. [28] developed multiple hardware accelerators and integrated the XMSS-specific hardware accelerators into a RISC-V processor. The resulting software-hardware co-design of XMSS works for trees containing 2^{10} key pairs and achieves a significant speedup compared to a pure software version. As shown in Table 3 despite a lower FMax, this work achieves considerably better performance for key generation than our general-purpose objective. For the purpose of analysing the time-area product, a detailed comparison between

[5] `https://github.com/joostrijneveld/xmss-reference`, commit fb7e3f8
[6] `https://github.com/intel/tinycrypt`, commit 5969b0e
[7] `https://bench.cr.yp.to/supercop.html`, version 20190910
[8] crypto_hash/keccakc512/simple/
[9] `https://github.com/riscv/riscv-gnu-toolchain`, commit 2c037e6

Table 2: Comparison with related work.

	Parameters (n,H,w)	Hash function	Platform	FMax (MHz)	KeyGen[a]	Sign[a]	Verify[a]
Ghosh et al. [11]	256,16,16	Keccak-400	Q+Stratix IV	32	—	4.8	4.8
our work[b]	256,10,16	SHA-3	Murax SoC	105	2766	6.4	5.9
Wang et al. [28]	256,10,16	SHA-256	Murax SoC	93	320[c]	0.93[c]	0.54[d]
our work[b]	256,10,16	SHA-256	Murax SoC	106	3645	7.9	4.7

[a] All results are given in 10^6 clock cycles.
[b] Our work running on Murax SoC with hardware accelerator.
[c] Based on the "Murax + Leaf + PRECOMP" design of [28].
[d] Based on the "Murax + Chain + PRECOMP" design of [28].

our results on the SHA-256 hardware implementation and the results from Wang et al. [28] is presented in Table 4. Despite the more general approach, Table 4 shows that our software-hardware co-design achieves a better performance and a better time-area product in signing and verification.

Table 3: Comparison of hardware and software implementation on a VexRiscv.

	KeyGen[a]		Sign[a]		Verify[a]	
	SHA-256	SHA-3	SHA-256	SHA-3	SHA-256	SHA-3
Software	92326	242784	131.5	356.3	49.9	120.3
Hardware	3645	2766	7.9	6.4	4.7	5.9
speedup factor	25x	87x	17x	55x	10x	20x

[a] All results are in 10^6 clk cyc. for $n = 256$, $w = 16$, and $H = 10$ (average of 100 iterations).

6 Conclusion

The presented work focuses on hardware and software implementations of hash functions on a RISC-V processor, and on different instantiations of XMSS. Following the approach of modularising and sharing components with a high utilisation in hash-based signature schemes, we integrated the components for hash operations as an instruction-set extension on the RISC-V architecture as atomic as possible. We chose hash functions following different design principles to evaluate the generality of our instruction set extension interface as well as the performance impact from architectural point of view. In detail, we compared SHA-2, following the Merkle-Damgård design principles, and SHA-3 as a sponge-based hash function.

Further, we investigated the impact of different implementations of hash-functions with respect to the integration via an ISA. Thus with the reduced latency the performance of the individual hash functions could be compared due to rounds and execution time, etc. In perspective of flexibility on long-term solutions, also known as crypto-

Table 4: Comparison of the time-area product of the SHA-256 hardware implementation with Wang et al. [28] for $n = 256$, $w = 16$, and $H = 10$.

		FMax (MHz)	Time	Area (slices)	Time x Area[a] (ratio)
key gen	**Wang et al.**[b]	**93.1**	**3.44** s	**14260**	**1.00**
	Wang et al.[c]	95.2	7.80 s	11328	1.80
	This work	106	34.38 s	5759	4.03
sign	Wang et al.[b]	93.1	9.95 ms	14260	3.30
	Wang et al.[c]	95.2	17.8 ms	11328	4.69
	This work	**106**	**7.45 ms**	**5759**	**1.00**
verify	Wang et al.[b]	93.1	5.80 ms	14260	3.24
	Wang et al.[c]	95.2	5.68 ms	11328	2.52
	This work	**106**	**4.43 ms**	**5759**	**1.00**

[a] Based on Time and Area relative to the respective most efficient design (bold)
[b] Based on the "Murax + Leaf + PRECOMP" design [28]
[c] Based on the "Murax + Chain + PRECOMP" design [28]

agility together with the drawback of a fixed instruction set via the ISA, a hybrid approach might also be of interest. Is it possible to merge two hash-algorithms in an implementation with shared resources with a general ISA API to be more future proof than using only one hash algorithm without sacrificing the efficiency in terms of resources and throughput? In contrast to [28], which focuses on XMSS specific optimisations, the aim of our work is a more generic optimisation of hash functions in order to enable post-quantum cryptography in today's devices. Therefore, the results of this work can also be used for other approaches that have a high utilisation of hash operations such as lattice-based signature schemes. Kannwischer et al. [17] provide a very comprehensive evaluation of hash operation utilisation of various lattice schemes. Table 3 shows that our generic hardware accelerator achieves a significant speedup without specific adaptations for XMSS. In an embedded environment usually the procedures for key generation and signing take place in a resource-rich backend, while signature verification is processed on a embedded device. Hence, implementations for signature verification shall focus on speed and area to optimise the cost. As presented in Table 4, in case of signing and verification our SHA-256 implementation only requires 40% (50% resp.) of the area compared to the fastest design of [28], and thereby achieves a better time-area product. According to our results, in the case of XMSS general-purpose hardware accelerators for hash functions are the more suitable solution in an embedded environment due to their reusability, the smaller required area, and the achieved performance.

Acknowledgments. We would like to thank Viola Campos, Thorsten Knoll, Michael Meyer, Peter Schwabe, and Wen Wang for their valuable contribution.

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 **2018**(1), 18–39 (Feb 2018), https://tches.iacr.org/index.php/TCHES/article/view/831
2. Bernstein, D.J.: Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete? SHARCS **9**, 105 (2009), https://cr.yp.to/papers.html#collisioncost
3. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering **2**(2), 77–89 (2012)
4. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ signature framework. In: Conference on Computer and Communications Security–CCS '19. Ed. by XiaoFeng Wang and Jonathan Katz. To appear. ACM. pp. 17–43 (2019)
5. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: Latin American Symposium on Theoretical Informatics. pp. 163–169. Springer (1998)
6. Buchmann, J., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the security of the Winternitz one-time signature scheme. In: Progress in Cryptology - AFRICACRYPT 2011. pp. 363–378. Springer (2011)
7. Buchmann, J., García, L.C.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS - an improved Merkle signature scheme. In: International Conference on Cryptology in India. pp. 349–363. Springer (2006)
8. Buchmann, J.A., Dahmen, E., Hülsing, A.: XMSS - A practical forward secure signature scheme based on minimal security assumptions. In: Yang, B. (ed.) PQCrypto. pp. 117–129. Springer (2011)
9. Dods, C., Smart, N.P., Stam, M.: Hash based digital signature schemes. In: IMA International Conference on Cryptography and Coding. pp. 96–115. Springer (2005)
10. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory (1985)
11. Ghosh, S., Misoczki, R., Sastry, M.R.: Lightweight Post-Quantum-Secure Digital Signature Approach for IoT Motes. Cryptology ePrint Archive, Report 2019/122 (2019), https://ia.cr/2019/122, version: 20190213:033538
12. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing. pp. 212–219. STOC '96, ACM, New York, NY, USA (1996)
13. Hülsing, A.: W-OTS+–shorter signatures for hash-based signature schemes. In: Progress in Cryptology - AFRICACRYPT 2013. pp. 173–188. Springer (2013)
14. Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. Internet Research Task Force (IRTF), 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. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International journal of information security **1**(1), 36–63 (2001)
17. Kannwischer, M.J., Rijneveld, J., Schwabe, P.: Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In: International Conference on Applied Cryptography and Network Security. pp. 281–301. Springer (2019)
18. van der Laan, E., Poll, E., Rijneveld, J., de Ruiter, J., Schwabe, P., Verschuren, J.: Is java card ready for hash-based signatures? In: International Workshop on Security. pp. 127–142. Springer (2018)

19. Lamport, L.: Constructing digital signatures from a one-way function. Tech. rep., Technical Report CSL-98, SRI International Palo Alto (1979)
20. McGrew, D.A., Curcio, M., Fluhrer, S.R.: Leighton-Micali Hash-Based Signatures. RFC **8554**, 1–61 (2019). https://doi.org/10.17487/RFC8554, `https://doi.org/10.17487/RFC8554`
21. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Advances in Cryptology - CRYPTO' 87. pp. 369–378. Springer (1987)
22. Merkle, R.C.: A certified digital signature. In: Conference on the Theory and Application of Cryptology. pp. 218–238. Springer (1989)
23. National Institute of Standards and Technology: FIPS PUB 180-2: Secure Hash Standard. Federal Information Processing Standards Publication 180-2. National Institute of Standards and Technology (2002)
24. National Institute of Standards and Technology: standard: Permutation-based hash and extendable-output functions. Draft FIPS 202, 2014 (3)
25. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2), 120–126 (1978)
26. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Robshaw, M., Katz, J. (eds.) Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on, pp. 124–134 (1994)
27. Shoufan, A., Huber, N.: A fast hash tree generator for Merkle signature scheme. In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 3945–3948. IEEE (2010)
28. Wang, W., Jungk, B., Wälde, J., Deng, S., Gupta, N., Szefer, J., Niederhagen, R.: XMSS and Embedded Systems - XMSS Hardware Accelerators for RISC-V. Cryptology ePrint Archive, Report 2018/1225 (2018), `https://ia.cr/2018/1225`, version: 20190522:113021
29. Waterman, A., Lee, Y., Patterson, D., Asanovic, K.: The RISC-V Instruction Set Manual. Volume I: User-Level ISA, version (2014)
30. Weber, A., Reith, S., Kasper, M., Kuhlmann, D., Seifert, J.P., Krauß, C.: Sovereignty in information technology (2018), `https://www.itas.kit.edu/pub/v/2018/weua18a.pdf`