Journal of Cryptographic Engineering
https://doi.org/10.1007/s13389-024-00362-4

RESEARCH ARTICLE

Extended version—to be, or not to be stateful: post-quantum secure boot using hash-based signatures

Alexander Wagner[1,2] · Felix Oberhansl[1] · Marc Schink[1,2]

Received: 22 June 2023 / Accepted: 20 July 2024

Abstract
While research in PQC has gained significant momentum, its adoption in real-world products is slow. This is largely due to concerns about practicability and maturity. The secure boot process of embedded devices is one scenario where such restraints can result in fundamental security problems. In this work, we present a flexible hardware/software co-design for HBS schemes which enables the transition to a post-quantum secure boot today. These signature schemes stand out due to their straightforward security proofs and are on the fast track to standardisation. Unlike previous work, we exploit the performance intensive similarities of the stateful LMS and XMSS schemes as well as the stateless SPHINCS$^+$ scheme. Thus, we enable designers to use a stateful or stateless scheme depending on the constraints of each individual application. To demonstrate the feasibility of our approach, we compare our results with hardware accelerated implementations of classical asymmetric algorithms. Further, we outline the use of different HBS schemes during the boot process. We compare different schemes, show the importance of parameter choices, and demonstrate the performance gain with different levels of hardware acceleration.

Keywords Post-quantum cryptography · Hash-based signatures · LMS · XMSS · SPHINCS$^+$ · Secure boot · Hardware/software co-design

1 Introduction

The boot process plays an important role to guarantee the security and trustworthiness of modern electronic devices. The first piece of software that is executed is stored in read-only memory (ROM). This boot code is the first step of a process called *secure boot* which ensures that only trusted and genuine software is executed from the very beginning. The importance of this role combined with the inability to update requires foresighted design decisions and carefully developed software. This is especially relevant, but not limited, to the used cryptographic primitives. Today's state-of-the-art implementations rely on common asymmetric algorithms like RSA or ECC [47, 51]. Through the development of quantum computers these algorithms are at risk as attacks based on Shor's algorithm may become feasible in the future [67].

In order to prepare for this threat, a process to standardise quantum-resistant public key cryptographic algorithms was initiated by the National Institute of Standards and Technology (NIST) in 2016 [55]. At the end of the third round, NIST selected the stateless hash-based signature (HBS) scheme SPHINCS$^+$ for standardisation [53]. It is the only selected algorithm not relying on the security of structured lattices. Stateless schemes can be used in the same manner as common digital signature algorithms based on RSA andECC. In contrast, stateful schemes require the signer to keep track of the already used keys as only a limited amount of signatures can be generated per key pair [50]. Any failure to do so seriously degrades the security [35]. The advantage of stateful schemes over stateless schemes is the smaller signature size and faster runtime. For the two stateful HBS schemes,

Alexander Wagner, Felix Oberhansl and Marc Schink have contributed equally to this work.

✉ Alexander Wagner
alexander.wagner@aisec.fraunhofer.de

Felix Oberhansl
felix.oberhansl@aisec.fraunhofer.de

Marc Schink
marc.schink@aisec.fraunhofer.de

1 Fraunhofer AISEC, Garching near Munich, Germany

2 Technical University of Munich, Munich, Germany

Published online: 13 August 2024

Leighton-Micali Hash- Based Signature (LMS) and extended Merkle signature scheme (XMSS), IETF RFCs are available [36, 49]. Based on these documents, HBS published a recommendation for the use of stateful HBSs in 2020 [19]. In 2021, the BSI extended their technical guidelines to include stateful HBSs. In 2022, recommendations for deployment of HBSs were published by the ANSSI. The soundness of HBSs relies only on the properties of the underlying hash functions. As hash functions are well understood, this makes HBS schemes a very conservative choice, in particular when compared to other post-quantum cryptography (PQC) algorithms [11, 48]. Due to this and their maturity, hybridation is not required [5, 14]. This makes HBSs a perfect fit for secure boot.

While maturity is not an area of concern for HBS schemes, practicability is. In case of secure boot, the startup time and accordingly the signature verification is of major importance. The verification time of hash-based signatures is mostly determined by the underlying hash function. To enable HBS schemes for secure boot, we propose a hardware/software co-design with minimal additional hardware overhead. Thus, allowing an immediate drop-in replacement of hash hardware accelerators. For evaluation purposes, we integrate our hardware accelerator in the OpenTitan. OpenTitan is a reference design for open source security controller and is based on a 32-bit RISC-V processor. We use the secure boot process of the OpenTitan to compare our implementations against the existing hardware-accelerated signature verifications based on RSA and ECC. Further, we lay out why a singular focus on either stateful or stateless schemes is a blocker for real world applications. In short, this is due to the fact that for most products different constraints are applicable. In the context of the secure boot, this is even more evident, as the successive boot stages mean that different entities and their respective constraints are involved. We overcome this issue with our flexible co-design accelerating stateless as well as stateful schemes from boot up.

In recent years, stateful schemes were evaluated for a usage in secure boot [43, 45] or for general purpose usage with efficient implementations [16, 34, 68, 71]. These implementations range from software evaluations including comparisons of different schemes [16, 42] to System-on-Chips (SoCs) with different levels of hardware acceleration [34, 43, 71] and full hardware designs [45, 68].

A flexible HBS solution for secure boot is still missing from state-of-the-art literature. Specifically, our paper provides the following contributions:

- A flexible hardware accelerator with support for stateful and stateless schemes, namely LMS, XMSS and SPHINCS$^+$ (also referred to as SPX$^+$ in the following)
  - Modular design approach to enable different levels of acceleration
  - Utilisation of algorithmic similarities to achieve low hardware overhead
  - Drop-in replacement/extension for generic hash cores
- Evaluation of algorithmic accelerations and trade-offs for hardware/software co-designs
- Exploration of HBS parameter trade-offs for the secure boot scenario
- Benchmark results in the context of secure boot with respect to code size, signature and key size, speed and area requirements

2 Hash-based signature schemes

Hash-based signatures are digital signature schemes that use hash functions as the main cryptographic primitive. The way how and which of these cryptographic primitives are combined allows to build either a so-called *stateful* or *stateless* HBS scheme. The usage of a stateful HBS algorithm differs for the signer compared to common asymmetric cryptographic algorithms. In essence, the number of signatures that can securely be generated is limited by the total number of key pairs available. Each key pair can only be used once and any reuse would result in a security compromise [35]. Thus, the signer has to keep track of the already used key pairs, effectively storing a state and updating it after every signature generation, i.e. stateful [50]. Contrarily the usage of stateless HBSs is in line with classical asymmetric cryptographic algorithms.

Aside from stateful and stateless, HBS schemes can be divided into the two variants, “simple” and “robust”, for which different security arguments in relation to the used hash function can be made. The “simple” instantiations have a less conservative security argument but a better performance. In contrast, “robust” instantiations meet more conservative security requirements and consequently require more hash operations. The reader is referred to [19, 39] for more information on the security arguments and its details.

2.1 One-time signatures

The foundation for contemporary HBS algorithms are the one-time signature (OTS) schemes. LMS, XMSS and SPHINCS$^+$ each use variants of the Winternitz OTS (WOTS). The core idea is to have a certain amount of function chains, i.e. repeatedly applying a function F to the prior output. For the sake of convenience, we assume that the function F only consists of a single call to a cryptographic hash function with the prior output as single input. Thus, in the following

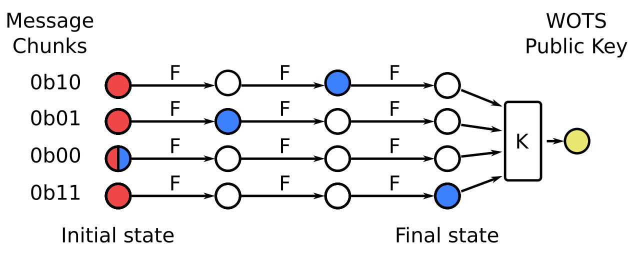

Fig. 1 The Winternitz OTS with the secret key ●, the compressed public key ●, and the signature ● (color figure online)

we simply use the term hash chain for this specific construct. The working principle of such OTS schemes is depicted in Fig. 1. The starting point of a hash chain is a random value and corresponds to one OTS secret key ●. An intermediate value of a hash chain is an OTS signature ●. The end point of a hash chain is an OTS public key. The function K is applied to these end points to generate the compressed OTS public key ●. For the compression function K, and SPHINCS$^+$ use a tweakable hash function and XMSS a so-called L-tree. The signing and verifying operations are inherently similar. To sign or verify a message, its digest is split into chunks of $log_2(w)$ bits and each chunk is interpreted as value a. For **signing** ●→● and **verifying** ●→● each hash chain is advanced. For signing it is advanced by a and for verifying by $w-1-a$. In the case of verification, a signature is valid, if the public key candidate is equal to the public key.

The function chain length and the chunk bit-size is defined by the Winternitz parameter w and $log_2(w)$, respectively. In the example in Fig. 1, the message is split into chunks of 2bit which corresponds to a Winternitz parameter w of 4. For more details with respect to implementation the reader is referred to [31].

2.2 Few-time signatures

In contrast to the OTS scheme a few-time signature (FTS) scheme allows the reuse of a key pair for a few times. The FTS scheme is only used in the stateless HBS scheme SPHINCS$^+$. As a result the total tree height of SPHINCS$^+$ can be reduced significantly, making it applicable in practice [9]. In the context of SPHINCS$^+$, the forest of random subsets (FORS) scheme is used to sign message digests. FORS consists of k Merkle trees each having a tree height of a. To generate the FORS public key ● all k Merkle tree root nodes are compressed with a hash. A single tree authenticates $t = 2^a$ FORS secret keys ●. Thus, the leaves are the hashes of the secret keys. To generate a signature the message digest is split into k a-bit chunks, as shown in Fig. 2. Each chunk is interpreted as an integer which is used as an index to select a secret key as signature node ●. This is done for all k trees and chunks. The selected nodes are aggregated together with the respective authentication path nodes ●. For verification the

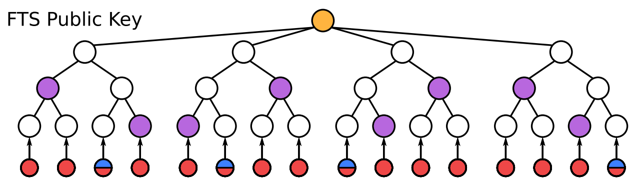

Fig. 2 Overview of the forest of random subsets scheme with the secret key ●, the public key ●, the signature ●, and the authentication path nodes ● (color figure online)

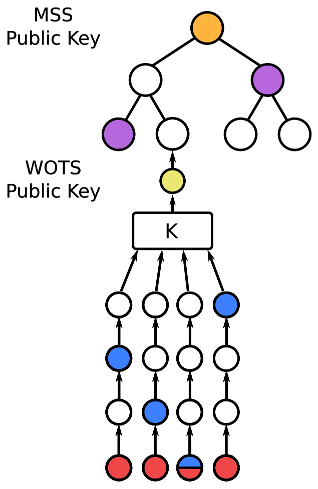

Fig. 3 Merkle signature scheme

signature is used to generate a public key candidate similar to the WOTS signature verification [39]. To allow for comparison, the same message is signed in the OTS and the FORS example, depicted in Fig. 1 and Fig. 2, respectively.

2.3 Merkle signature scheme

With an OTS or FTS scheme, one key pair can be used once or a few times to sign, respectively. In order to overcome this limitation, the Merkle signature scheme (MSS) is used, depicted in Fig. 3. It applies a Merkle tree to authenticate multiple OTS public keys. Every leaf node in the Merkle tree corresponds to a single hashed OTS public key. The root node of the tree corresponds to the MSS public key, which is used to authenticate the OTS public keys. A Merkle tree with a tree height h authenticates 2^h OTS key pairs [52].

A MSS signature consists of an OTS signature, introduced in Sect. 2.1, and, in the context of HBS, a so called *authentication path*. Starting at the bottom of the figure, the OTS public key ● is calculated from the signature ●. Then the authentication path nodes ● are used to generate the public key candidate ● for the respective signature. The signature is valid, if the public key candidate is equal to the known public key.

With the MSS an OTS can be extended to a many-time signature (MTS). Such a HBS construct is applicable to real

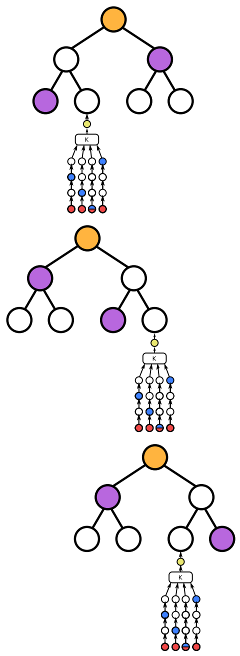
Fig. 4 Generalized Merkle signature scheme

world use cases, but still impractical for a high number of key pairs, i.e. required signatures. The tree height h is limited by the runtime of key generation and signing. To overcome this limitation the generalized Merkle signature scheme (GMSS) was introduced in [15]. Its core idea is to build a so-called certification tree with multiple MSS. Instead of having a single MSS with a large Merkle tree, it is split into d MSS each being a smaller Merkle tree. At the top of this certification tree is the root MSS, which signs its child MSS. At the bottom is the leaf MSS, which is used to sign the message.

For SPHINCS$^+$, GMSS is especially relevant, as the total required tree height h is vast [39]. Different to the exemplary overview in Fig. 4, SPHINCS$^+$ uses a FTS scheme instead of an OTS scheme in the bottom layer, as explained in Sect. 2.2.

3 Secure boot with hash-based signatures

In this section, we describe the different stages and involved entities of a secure boot process. For signing entities, we give recommendations for efficient state management that is compliant with NIST SP 800-208. The relevance of this specification is due to the recently released Commercial National Security Algorithm Suite 2.0 [69]. From 2025 onwards, new national security systems of the United States of America must perform software and firmware signing with HBSs. The transition of all deployed software and firmware signing must be completed until 2030. The deployed signature algorithms must meet all requirements of the NIST SP 800-208. This enforces the use of NIST SP 800-208 and makes this publication highly relevant for any vendor. For the verifying entity, we focus more on implementation aspects, to enable efficient post-quantum secure boot on embedded devices. To this end, we show how real world applications benefit from our hardware/software co-design and its flexibility to support stateful and stateless signature schemes. Further, we derive parameters for all considered HBS schemes tailored for the secure boot use case.

A secure boot process consists of multiple stages which are executed one after the other until the application is started. To ensure that only genuine software is executed, every boot stage verifies the next stage before it hands over execution. Each stage is associated with an individual role belonging to a different entity. The stages and roles are based on the specification of the boot process for the OpenTitan project, but apply to most use cases. The involved roles in the boot process are *silicon creator*, *silicon owner*, and *provider*.

The *silicon creator* is responsible for the first stage of the boot process and thus the root of trust and security for the entire device. As the software for this boot stage is stored in ROM, it cannot be updated after the chip production. Thus, foresighted decisions with respect to the chosen signature scheme and a secure implementation are of utmost importance. The *silicon owner* is the entity that uses the hardware and provides, for example, the kernel or operating system (OS) for the silicon. This boot stage is usually stored on an updateable non-volatile memory. The application of the product is the last stage of the boot process. The *provider* is responsible for this boot stage.

Each entity is responsible to provide valid signatures for their respective boot stages. Thus, the key generation, maintaining the key material and signing needs to be handled individually by each entity. Depending on the constraints and capabilities each entity may decide to use stateful or stateless HBSs.

3.1 Distributed state management compliant with NIST SP 800-208

If an entity decides to use stateful HBSs, it is necessary to prepare for any hardware failure regarding the hardware modules holding the private key and its state. A common approach would be to copy this data and distribute it to multiple hardware modules. To avoid any location-specific single point of failure, these devices would be located at different physical locations to mitigate the threat of local hazardous event. For stateful HBSs the recommendations by NIST prohibit any export of private key material [19]. Hence, a straightforward distribution is not possible feasible. The NIST recommendations outline alternatives, which we summarises in the following. In addition, we extend the—in our opinion—most favorable approach and describe how the internal structure of HBSs can be used to enable significantly faster verification and smaller signatures.

Recommendations of NIST SP 800-208

One option presented in the NIST recommendations proposes to generate multiple key pairs and register these within the device at once or to allow for an update of the public key. A second option corresponds to the usage of a GMSS, such that a top-level and multiple bottom-level trees exist. The bottom-level trees are signed by the top-level tree. Each bottom-level tree can be stored in a different hardware module without exposing any secret key material. Hence, the capability to create signatures for one public key is distributed among different modules, without the necessity of exposing secret key material.

Distribution of sub-trees

We deem the second approach most feasible in practice, as it does not rely on any features within the device that performs verification. This approach relies on a multi-tree signature with a distinct division into one top and multiple bottom-level trees. This division doubles the signature size and verification runtime, as an application signature must contain one signature for the bottom-level tree and one signature to verify this signature with the top-level tree. We suggest an improvement to allow for a single-tree signature by using the internal HBS structures instead. The change targets to introduce sub-trees within one tree. Each sub-tree is handled by an individual hardware module. During the key generation, the public key is calculated jointly by all hardware modules. After the key generation phase, each hardware module is able to independently generate valid signatures.

In Fig. 5, this approach is shown for a simple tree with four sub-trees. During key generation, each hardware module generates its sub-tree leafs as it would if generating the complete single tree. For instance, the first hardware module generates *sub-tree a* and calculates all nodes up to and including node four in the merkle tree. The nodes of the top-level tree, in our example the nodes with the index one, two, and three, cannot be calculated by an individual hardware module. For this, the four hardware modules need to cooperate. Each hardware module must publish its respective sub-tree root node, such that the other hardware modules can use this as an input for the generation of the top-level tree. After the initial key generation, each module must be able to generate signatures independently to ensure redundancy. Therefore, each hardware module must store the nodes that are needed for the authentication path of signatures generated from its sub-tree. For example, the hardware module containing *sub-tree a* has to store the nodes five and three internally. With these values, this hardware module is able to independently generate signatures in the future. The same applies to the other hardware modules each handling a different sub-tree.

This idea is similar to the signature generation acceleration with auxiliary data that can be used for LMS [18] and was also proposed for SPHINCS$^+$ [37]. The nodes contained

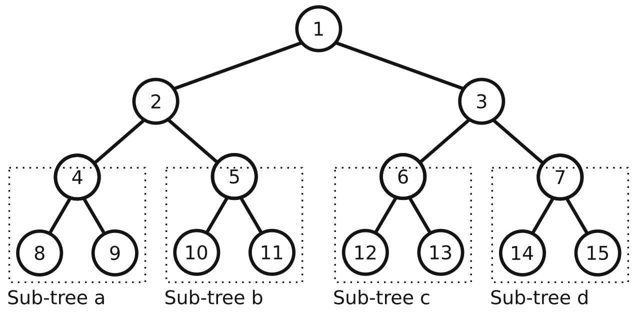

Fig. 5 Split of a single tree with height three into four sub-trees each containing two one-time signature key pairs

in the auxiliary data may be stored internally along with the private key or externally. If it is stored externally it must be cryptographically protected against modifications. Any modification would result in an invalid signature, and thus in a denial-of-service. The LMS implementation appends an HMAC to the auxiliary data. Confidentiality is not an issue for the authentication nodes, as nodes within the merkle tree are public data, therefore they can be stored as plaintext. A signature contains certain nodes of the merkle tree as authentication path to enable the calculation of public key candidate. Therefore our proposed sub-tree approach is uncritical from this perspective. As no private key material is exported, it is compliant with the NIST SP 800-208 [19].

The major benefit of our proposed sub-tree optimisation is that it only requires a single-tree in contrast to an approach with a multi-tree structure. Therefore, only one signature needs to be verified, which increases performance and reduces signature size. The overhead of our state management is reduced to the overhead that is inherently given by the need for redundancy, i.e. the overhead in signature size and performance is defined by the "parent" section of the tree that connects all sub-trees (nodes one to three in Fig. 5). The number of layers in the parent section defines the number of redundant modules that are available. These layers increase the signature size and verification time. However, their impact is minimal, as only a single node in the authentication path per parent layer is required. A drawback of this approach is that it requires a dedicated mode within HBS libraries. This is due to the fact, that the key generation and signing operations slightly differ from a non-distributed variant. Hence, this requires some software engineering effort.

Our proposed state management solution does not allow or require any updates or changes in the future. Hence, all keys must be generated during the initial key generation phase.

3.2 To be, or not to be stateful

Stateful HBS schemes allow for fast verification with small signatures. The drawback of a stateful HBS scheme is the requirement to maintain its state. Best practices for state management were evaluated in [50]. In essence, three different approaches were proposed and outlined. Two of them requiring dedicated cryptographic hardware, like hardware security modules (HSMs), which are capable of securely synchronising the state. The third approach uses a combination of a stateful and a stateless HBS scheme. This approach does not require any cryptographic hardware, but has the drawback that the verifier must perform a stateless and a stateful signature verification. Hence, this is not of general interest for the secure boot use case. In general, state management is expensive and any implementation must guarantee high assurance. Depending on the respective application, entities cannot generally bear this overhead. Thus, the question on whether the HBS scheme should be stateful or not must be answered differently, depending on the application, but also the entity. Since every boot stage is controlled by an individual entity and *owner* and *provider* may have different constraints with respect to boot time and application security, some devices need to verify stateful signatures at one stage and stateless ones at another. Thus, our architecture that supports both types is the most promising approach to enable the transition to a post-quantum secure boot using hash-based signatures.

3.3 Choice of hash-based signature parameters

The general impact of the HBS parameters are formally described in [42]. Within the scope of this work, we select the parameters in accordance with the constraints of secure boot. The SPHINCS$^+$ submission document defines fixed parameter sets. This is in contrast to stateful schemes, where parameters can be selected more freely. In the following section, we provide an overview over the parameters and set forth the rationale behind our parameter choice.

Hash Algorithm LMS, XMSS and SPHINCS$^+$ have support for both SHA-2 and SHA-3. Due to the widespread use of SHA-2, we select the SHA-256 hash function with an untruncated output size of 32 bytes. This guarantees a level of security equivalent to an exhaustive key search for AES-256 [39], thus reaching NIST's highest security level five. If Grover's attack is feasible, this equals a level of 128 bits in a pre-quantum world. We select for ECC a 256-bit curve and for RSA 3072-bit integers (Table 1) for comparison with commonly used asymmetric algorithms.

LMS and XMSS

Due to the nature of stateful HBSs the ability to sign firmware images is limited to a fixed count, which is defined once at key generation (Sect. 2.3). For the secure boot use case we estimate the required number of firmware updates including a security margin. We estimate the maximum lifetime of a security controller to be 40 years and the amount of required updates to at most two updates each day during its lifetime. This totals to up to 29200 required signatures, where one signature is used for each firmware update. From this we can derive the required tree height parameter for LMS to be $h = 15$ and XMSS to be $h = 16$, with respect to the parameters listed in [19]. For XMSS, the Winternitz parameter w is limited to $w = 16$. For LMS, the Winternitz parameter is $W \in [1, 2, 4, 8]$, which maps to $w \in [2, 4, 16, 256]$. The notation of w is used for SPHINCS$^+$ and XMSS, so we will use this notation as well for LMS. As shown in Fig. 6, the Winternitz parameter allows a trade-off between signature size and overall performance. A higher Winternitz parameter generates smaller signatures, but has the drawback of worse performance. This is not the case for $w = 2$, as for both $w = 2$ and $w = 4$ the required hash compress calls add up to the same count, while the signature for $w = 2$ is larger. Therefore, the original Winternitz parameter set can be limited to $w \in [4, 16, 256]$. The resulting signature sizes of the selected parameters for LMS and XMSS are listed in Table 1.

SPHINCS$^+$

In contrast to XMSS and LMS, the SPHINCS$^+$ parameters are described with certain sets of parameter combinations. This is due to the fact, that the stateless property of SPHINCS$^+$ is a result of carefully combining parameters. Thus, we restrict our evaluations to the provided parameter sets. The available list of parameters can be split into the two variants, "small" and "fast", which are denoted with "s" and "f", respectively. The "small" variant has the drawbacks of slower key generation and signing. However, it achieves smaller signature sizes and faster verification. As this is desirable for the secure boot scenario, we select the "small" variant. The respective signature and public key size for the selected parameter set is listed in Table 1. The "simple" and "robust" construction that can be used in SPHINCS$^+$ only influence the security proof and runtime but not signature or public key sizes. They are referred to as SPX$^+$-s and SPX$^+$-r in the following.

4 Hardware/software co-design

To ensure the practicability of HBS schemes during runtime, we propose a flexible hardware/software co-design to enhance the performance of signature verification. In this chapter we lay out and evaluate our approach.

4.1 Software implementation

Within the following section, we lay out our design methodology starting with a peak into the general performance characteristics of LMS with $w = 16$, XMSS and SPHINCS$^+$. As depicted in Fig. 7, we differentiate the operations of the

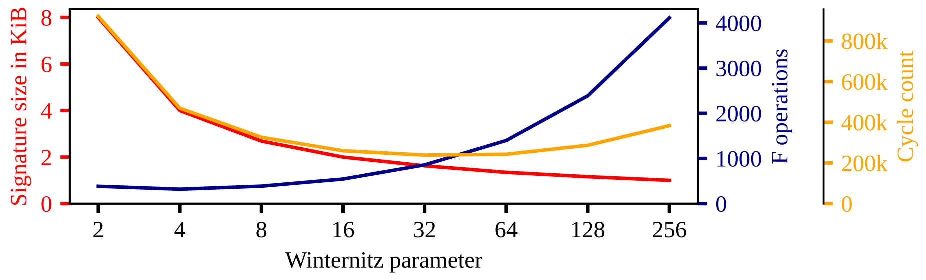

Fig. 6 Cycle count estimation for a signature verification of a WOTS accelerated by our hardware/software co-design depending on different Winternitz parameters and in comparison to the signature size and the required F operations

Table 1 Signature and public key sizes and the reached NIST security level, in comparison to RSA and ECC, for LMS, XMSS and SPHINCS$^+$ with SHA-256 as underlying hash function and an output size of 32 bytes

Algorithm	Parameter		Signature size	Public key size	NIST security level
LMS	$h = 15$	$w = 4$	4.7KiB	32B	5
	$h = 15$	$w = 16$	2.7KiB	32B	5
	$h = 15$	$w = 256$	1.6KiB	32B	5
XMSS	$h = 16$	$w = 16$	2.6KiB	32B	5
SPX$^+$	256s		29KiB	64B	5
RSA	3072		384B	384B	❦
ECC	$P - 256$		64B	64B	❦

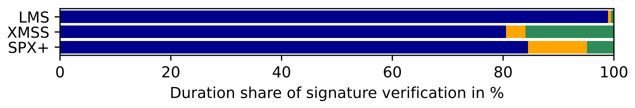

Fig. 7 Performance of software implementations impacted by hash chain, merkle tree, and other operations

algorithms into the three classes: hash chain, authentication path, and other. The other part was not further split as it takes up less than 10% on average. Hence, it is less important for hardware acceleration. The hash chain computation is responsible for over 80% of the overall latency during a signature verification. Therefore it is the most interesting part for acceleration by dedicated hardware. The authentication path takes up to 15% of performance for SPHINCS$^+$, making it a possible second target for acceleration.

Hash hardware accelerators

In general, hashing dominates the execution time of the HBS algorithms. Thus, any acceleration within the hash computation has a meaningful impact on the overall performance. SHA-2/3 hardware cores are found in many microcontrollers, as the algorithm is frequently required and their permutation logic can be implemented efficiently in hardware. The usage of such an accelerator shifts the bottleneck from computation to communication with the accelerator. On our target platform, the OpenTitan, a SHA-256 compress takes 65 clock cycles, while writing the data and reading the digest raises the latency to around 1400 clock cycles. For digesting a high amount of data this is irrelevant, as the transfer interleaves with computation and the compress function is executed multiple times. However, for a step within a hash chain 55 ((LMS) to at most 96 bytes (XMSS) are digested at once. Therefore, a generic hash accelerator is not ideal for usage in hash chains.

4.2 Hardware hash-based signature accelerators

Acceleration that can be achieved with generic hash cores is limited due to a high communication overhead. Due to the tree and chain structures in HBS schemes, data structs are often accessed subsequently. Dedicated hardware components can manage this data flow and consequently reduce interaction with the main processor. Therefore, we propose a set of HBS top modules that support the computation of a hash chain, the computation of a tree root, or both. We use the open source SHA-2/HMAC core from the OpenTitan project as hash backend. Throughout the rest of this paper we refer to our proposed design as SHA-2$^+$ core.

Winternitz parameter exploration for HW/SW Co-Designs

To assess the impact of our approach, we estimate the resulting cycle count for an OTS signature verification on our design in Fig. 6. As stated in Sect. 2, OTS verification consists mainly of advancing hash chains. The estimation shows that a hash chain module allows to use higher Winternitz parameters without significantly degrading the overall performance. As the chain length increases, the number of required I/O operations reduces, so runtime improves even if more hash operations are required. This is in direct contrast to software implementations where an increase of required F operations implies a linear increase in runtime. Interestingly, our estimate shows that a hardware/software co-design approach enables to reduce the signature size without affecting the performance. For example, a Winternitz parameter of $w = 256$

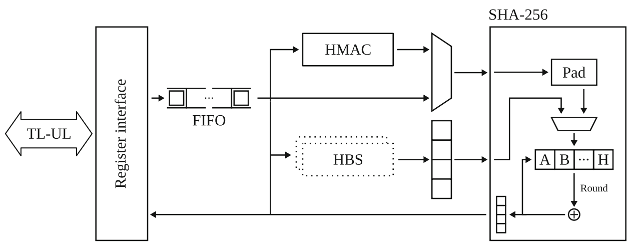

Fig. 8 Block diagram of the SHA-2$^+$ core

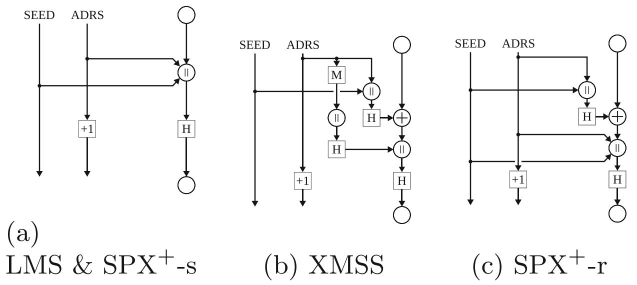

Fig. 9 Schematic depiction for a "simple" hash chain (LMS & SPX$^+$-s) and a "robust" hash chain in XMSS and SPX$^+$-r

can be selected instead of $w = 16$, to cut the OTS signature size into half without a significant performance regression.

***SHA-2$^+$* core**

Figure 8 shows our hardware design. The original OpenTitan SHA-256 accelerator consists of a SHA-256 backend, which can be accessed either transparently or through the HMAC toplevel, to compute a SHA-256 or a MAC, respectively. By reusing the SHA-256 logic, we minimise the additional cost for manufacturers. At synthesis time one of six HBS modules can be plugged into the design: (i) LMS, (ii) XMSS, (iii) SPX+-s, (iv) SPX+-r, (v) SPX+-s + LMS, (vi) SPX+-r + XMSS. In theory, these modules can be fitted for arbitrary hash cores with minor modifications. Our changes to the original hash accelerator include a chain register which is connected to the SHA-256's initial state register, additional control logic to switch between operation modes, and a digest feedback path into the HBS module. The HBS cores implement the respective behaviour by means of simple state machines. Our complete design can be directly integrated into any chip that supports the TileLink Uncached Lightweight (TL-UL) interface.

Combined stateful and stateless accelerators

The rationale for supporting both stateless and stateful signatures in one design was laid out in Sect. 3.2. The obvious choices for combination are LMS and SPX$^+$-s and XMSS and SPX$^+$-r, as they are respectively "simple" and "robust" instantiations of HBSs. Figure 9 shows the steps in a "simple" hash chain (LMS and SPX$^+$-s), a "robust" hash chain in XMSS, and a "robust" hash chain in SPX$^+$-r. The only difference between LMS and SPX$^+$-s is that the former uses 23 address bytes, whereas the latter uses 22 bytes and the initial SHA-256 state. In theory, both the "simple" and "robust" SPHINCS$^+$ variant would require to hash a public seed before each hash operation. This seed is padded such that this compress only needs to be done once and the resulting SHA-256 state can be reused for subsequent hash operations. All SPHINCS$^+$ cores support this behaviour. Apart from that, advancing a simple hash chain requires a single compress with an incrementing iterator and the output of the previous step. The differences in SPX$^+$-r and XMSS are more severe. WOTS$^+$ in XMSS requires to first compute a unique key and mask, where the hash function acts as a pseudorandom function, and finally to update the chain digest, for which the hash function acts as a keyed hash function. The SPX$^+$-r construct omits the calculation of a unique key. In XMSS, 32 address bytes are required, SPHINCS$^+$ uses 22 bytes. Therefore, in straightforward implementations, six compress iterations are required in XMSS, two each for the mask, key, and advancing the chain, and three compress iterations are required in SPHINCS$^+$, two for the mask, and one for advancing the chain. While the two "robust" instantiations in XMSS and SPHINCS$^+$ do not map as good as the two simple instantiations, the buffer registers and some control logic can be reused.

XMSS precomputation

The amount of compress iterations in an XMSS WOTS$^+$ hash chain step can be reduced via precomputation [71]. Three data structures are computed in a step, the mask, the key, and the data for the chain itself. For all three data structures, a public seed and an address must be compressed first. Since the public seed is constant, and the address is constant within a step, this compress only needs to be done once instead of three times. The resulting state can be used to overwrite the SHA-256 state in the subsequent hashes, thus lowering the number of compress iterations from six to four. This comes at the cost of a buffer register and overwrite logic for the SHA-256. As SPHINCS$^+$ requires this behaviour as well, this applies only to the standalone XMSS core.

Accelerating root computation in Merkle trees

In addition to accelerating hash chains, we also explore hardware optimisations for the *compute root* operation in Merkle trees. We limit this exploration to the FORS and MSS scheme in SPHINCS$^+$. Due to the usage of multiple tree levels in SPHINCS$^+$, more authentication path calculations than in LMS and XMSS are required (Fig. 7). Further, related work has shown that hardware features beyond hash chain computation offer little to no additional benefit for verification in XMSS [71], and as the performance of SPHINCS$^+$ lacks behind the stateful schemes in general, it is most relevant for additional acceleration. Therefore our SPX$^+$-s and SPX$^+$-r modules can be extended with a *MerkleTree* submodule. The *compute root* operation calculates the root of a tree from a leaf and the respective authentication path. Through all tree levels, two child nodes are combined to obtain their parent node with a hash function. In SPX$^+$-s, this corresponds to calculating one digest by compressing both child nodes with two compress iterations. For SPX$^+$-r, two compress iterations to obtain a 64 byte mask are required. Afterwards, the masked data is compressed within two more iterations. The order in which the nodes are hashed depends on whether the node from the authentication path is a left or right neighbour. Our core continuously absorbs nodes from the authentication path, reorders them with the node buffered in hardware and computes the hash to obtain the parent node. Only the root node is read from the SHA-2$^+$ core, thus dispensing all but one read operation.

Synthesis results

In Table 2, we report FPGA and ASIC synthesis results for all configurations of the SHA-2$^+$ core. As FPGA target, we chose an Artix-7, synthesis and implementation are done with default settings in Vivado 2020.2. For ASIC synthesis, we use the SG13s 130nm process by IHP and the Cadence Genus synthesis tool in version 21.10-p002_1. The OpenTitan SHA-256/HMAC core we use as basis consumes around 56.3kGE. The SHA-256 logic and its registers amount to slightly more than 50% of that. The next largest blocks are the register interface, the HMAC logic, and the FIFO, with approx. 20%, 13%, and 8%, respectively. We prove that integrating a HBS accelerator for LMS can be as cheap as 11kGE additional gates, an approximate overhead of 20%. Extending this to a design that also supports SPHINCS$^+$ costs an additional 5kGE, mainly due to the buffer register for the initial SHA-256 state. Additionally integrating the *MerkleTree* accelerator is comparatively expensive, as the behaviour differs fundamentally and registers to parse leafs and buffer a digest are required. The "robust" instances are more expensive, as mask and key registers are required. Integrating SPX$^+$-r requires no additional sequential logic, but 5kGE in combinatorial logic, as the differences in the hash chain step are more severe. Supporting the *MerkleTree* computation in the "robust" variant on top requires even more buffer registers and additional control logic. This becomes even more obvious if a standalone SPX$^+$-r core is built, as the hash chain core in SPHINCS$^+$ requires less registers than the XMSS core.

Our synthesised ASIC can be clocked at 125MHz at most. The target frequency of the OpenTitan is 100MHz. Even with the most complex SHA-2$^+$ synthesis configuration the critical path of the design is still determined by the SHA-256 round logic. The FPGA synthesis supports this finding, as the length of the critical paths deteriorates only minimally. On a medium-sized Artix-7 board with almost 50k LUTs, all our designs can be clocked at 100MHz.

Table 2 Synthesis results for different configurations of the SHA-2$^+$ core on FPGA on Artix7 and ASIC using the IHP 130 nm process SG13S [41]

	FPGA		ASIC		Overhead
	LUTs	FFs	Area	GE (mm^2)	(%)
SHA2, HMAC	3480	2400	0.319	56,300	–
+ LM-OTS	4400	3080	0.381	67,200	20
+ SPX$^+$-s	5060	3360	0.414	73,000	30
+ MerkleTree-s	5920	4070	0.485	85,600	52
+ WOTS+	4840	4000	0.464	81,800	45
+ SPX$^+$-r	5950	4000	0.492	86,800	55
+ MerkleTree-r	6210	4170	0.575	101,400	80
+ SPX$^+$-s	4870	3350	0.407	71,800	28
+ MerkleTree-s	5850	4050	0.476	84,000	49
+ SPX$^+$-r	5510	3860	0.458	80,800	44
+ MerkleTree-r	6050	4120	0.560	98,800	75

5 Benchmark results

We evaluate our flexible hardware accelerator in general and with respect to the secure boot use case defined in Sect. 3. As the verification time for hash-based signatures is not constant and depends on the signed message, we evaluate each parameter set and implementation with 1000 different messages. For a baseline, we start the comparison with our open source software implementations of LMS, XMSS and SPHINCS$^+$. Within this comparison all implementations are written in Rust and are not optimised for the RISC-V target architecture. We extend this by benchmarking the implementations accelerated by the available SHA-256 core as well.

5.1 LMS and XMSS

First, we evaluate the performance of signature verification for the stateful signature schemes LMS and XMSS. For this, we benchmark the verification in software running on the Ibex processor, accelerated by the general-purpose SHA-256 core, and accelerated by our SHA-2$^+$ core. The parameters of interest are listed in Table 1. The results are shown in Fig. 10.

For LMS, the relative performance improvements for our approach differ from the relative ranking of a software implementation with or without general-purpose SHA-256 core. Our results show that the benefit to use a hardware/software co-design depends on the Winternitz parameter as estimated in Fig. 6. Instead of exponentially increasing cycle counts for larger w, our architecture is fastest for $w = 16$. Signature verification with $w = 256$ performs only 25% slower than for $w = 16$ and even faster than for $w = 4$. Overall, signature verification with $w = 4$ is slowest and signature size is largest. Hence, we further restrict the Winternitz parameter for LMS to $w \in [16, 256]$.

Comparing the XMSS results with the LMS results for $w = 16$ leads to approximately the same relative performance increase. With our accelerator, the performance of XMSS signature verification is increased in comparison to software and the SHA-256 core by a factor of 12 and 4, respectively. Due to the "robust" construction, XMSS is slower by a factor of approximately 6. For LMS with $w = 256$, software and the SHA-256 core are outperformed by a factor of 80 and 20, respectively.

The relationship between different Winternitz parameters and performance for hardware/software co-design was never reported in detail. While it is straightforward for standalone hardware or software implementations (e.g. shown in [43]), co-designs require to consider costs for data transfers. For our design, the performance of LMS verification is effectively the same for all w, while the signature size is reduced by up to 50%. For XMSS this can be estimated to have even a bigger impact as the compression of the OTS public keys with K is performed by an L-tree. Increasing the Winternitz parameter from $w = 16$ to $w = 256$ effectively halves the required node operations within the calculation of the L-tree. As the L-tree generation is computationally expensive, halving the required operations impacts the overall performance. Therefore using Winternitz parameters $w > 16$ in XMSS would be desirable not only for secure boot, but as the standardisation is quite progressed, parameter changes are unlikely to happen.

Rapidly verifiable signatures

As it can be seen in Fig. 10, the execution of a signature verification is not performed in constant time. Based on this [13, 59] introduced a general algorithmic optimisation applicable for stateful HBSs. The approach does not reduce the total number of operations in signing and verifying but shifts computations from the verifier to the signer. The signer searches for a rapidly verifiable signature by appending a random counter to the message. If the resulting signature requires the verifier to perform less F operations to reach the final state, i.e. public key (see Fig. 1) in comparison to the original signature, a *rapidly verifiable* signature is found. This is repeated for a distinct number of trials. The random counter that requires the verifier to do the least F operations is used to generate a *rapidly verifiable* signature. For software implementations it has been proven that the performance benefits significantly and thus signer's additional effort is well spent. In the following section, we extend our implementations to give an insight into the impact of this optimisation on hardware/software co-designs. Further, we extend the existing knowledge base by applying this optimisation technique to LMS, as this was not yet performed.

The results for the rapidly verifiable signatures are plotted in Fig. 11. Performance improvements are evaluated up to a maximum optimisation counter equal to 10^9. For LMS and XMSS with $w = 16$ the relative performance improvement is comparable. For our software implementation and the acceleration with the general-purpose SHA-256 core, the optimisation decreases the runtime for signature verification by approximately 30%. In contrast, the runtime of the implementations accelerated by our SHA-2$^+$ core is only decreased by 10%. For LMS with $w = 256$ comparable observations can be made as the software and the SHA-256 variants have a decrease in runtime by 50% and the SHA-2$^+$ variant by 30%. In general, the optimisation reduces the runtime. However, the relative improvement is reduced for our hardware/software co-design. This is due to the fact that advancing within the hash chains with our design is not as expensive compared to the rest of the algorithm.

Further, we extend the results reported by [13] for Winternitz parameters $w > 16$ with our benchmark results. A larger Winternitz parameter allows for bigger improvement of the performance. It can be seen that the benefit of using the rapidly verifiable approach for LMS and XMSS with $w = 16$, accelerated by our SHA-2$^+$ core, is neglectable and the effort may not be required by the signer. For larger Winternitz parameters of $w = 256$ it still is a decent improvement of the performance.

5.2 SPHINCS$^+$

The performance of the SPHINCS$^+$ signature verification was evaluated similar to the stateful HBS benchmarks in software, accelerated by a SHA-256 accelerator and by our SHA-2$^+$ core (provide link for figure 12). As presented in Sect. 4, our hardware accelerator comes in two different variants for the acceleration of SPHINCS$^+$. The basic SHA-2$^+$ core accelerating the hash chain calculation and the extended SHA-2^{++} core including the *MerkleTree* module accelerating the *compute root* calculation in Merkle trees as well. The SHA-2$^+$ core speeds up the signature verification for "simple" and "robust" by approximately a factor of 14. Similar to the stateful HBS benchmarks the execution time varies. The SHA-2^{++} core improves the performance compared to a software implementation for "simple" and "robust" by roughly a factor of 21 and 27, respectively. With respect to the SHA-2$^+$ core the extended SHA-2^{++} reduces the latency by 34% for "simple" and 45% for "robust". Considering the relative high hardware utilisation listed in Sect. 4 makes the extended core only suitable for scenarios with strict timing requirements. As our evaluation in Sect. 4 has shown, further hardware accelerations are not suitable as the latency is not heavily influenced by the not yet accelerated calculations. This makes our SHA-2$^+$ core a very efficient implementation

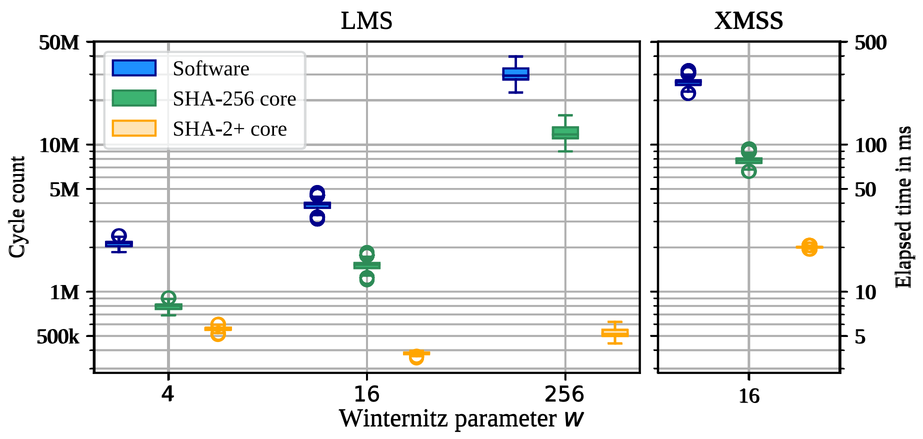

Fig. 10 Signature verification time for 1000 different messages with LMS and XMSS. Both are implemented in software and accelerated by a SHA-256 and SHA-2^{+} core

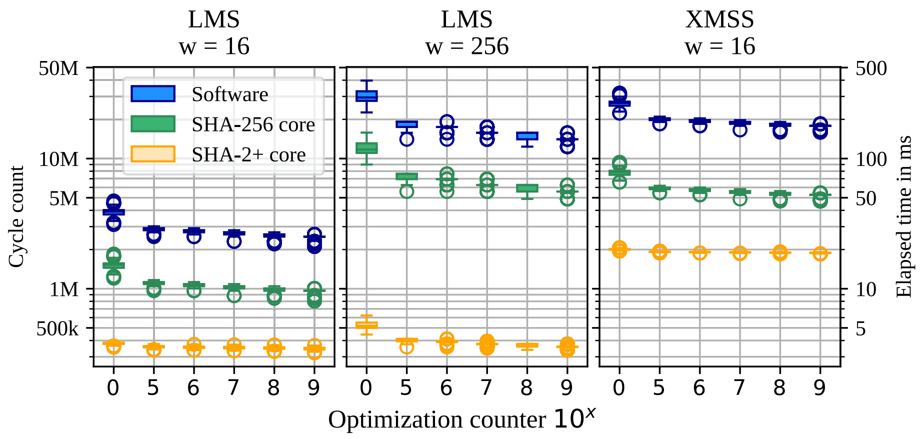

Fig. 11 Signature verification time for 1000 different messages with LMS and XMSS. Optimisation with the *rapidly verifiable* approach with different optimisation counters. Both algorithms are implemented in software and accelerated by a SHA-256 and SHA-2^{+} core

Table 3 State-of-the-art HBS implementations

	Algorithm	Approach	Area FPGA		Performance MHz	ms
[16]	LMS	SW	–		24	110
	$w = 16, h = 10$	ARM Cortex-M4				
[43]	LMS	FPGA	968	LUTs	250	6.3
	$w = 265, h = 15$		517	FFs		
			2	BRAMs		
			1	DSP		
[16]	XMSS	SW (rap. verif.)	–		24	273
	$w = 16, h = 10$	ARM Cortex-M4				
[71]	XMSS	HW/SW	2580	Registers	95.2	5.68
	$w = 16, h = 10$	RISC-V RV32IM	1700	ALMs		
[44]	SPX^{+}	SW	–		24	729
	$256s - simple$	ARM Cortex-M4				
[44]	SPX^{+}	SW	–		24	2070
	$256s - robust$	ARM Cortex-M4				

with low overhead and our SHA-2^{++} core more performant with the drawback of higher hardware costs.

Similar to benchmarks with LMS it would be interesting to evaluate the impact of a higher Winternitz parameter with $w = 256$. As shown with the LMS benchmarks, the performance can be expected to be constant. This is due to the fact that FORS, which is the main design difference, does not depend on the Winternitz parameter. The update in parameters would reduce the signature size from 29KiB down to 21KiB. We did not further investigate $w = 256$ for SPHINCS^{+} and leave it open for future work.

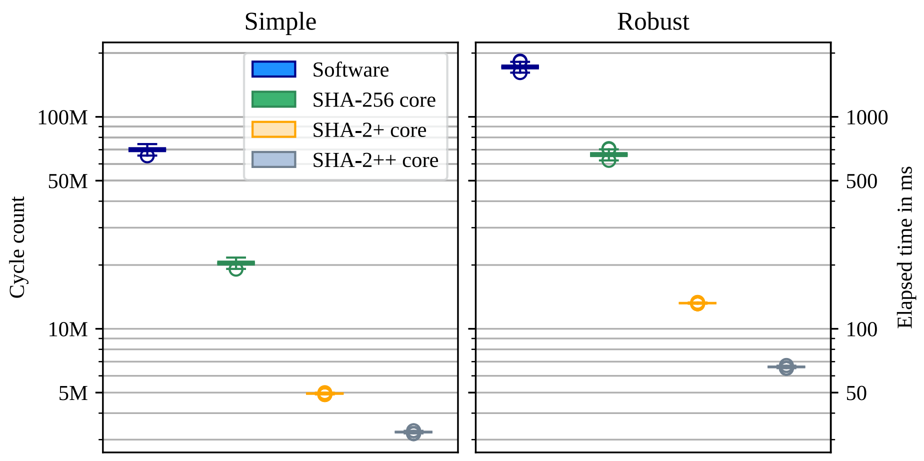

Fig. 12 Signature verification time for 1000 different messages with SPX$^+$-s and SPX$^+$-r implemented in software and accelerated by a SHA-256, our SHA-2$^+$ core, and our extended SHA-2$^+$ listed as SHA-2^{++}

5.3 Comparison with related work

Table 3 summarises the state of the art relevant to this work. The reported hardware utilisations do not include the SHA-256, but only the relative overhead for supporting HBSs Complete FPGA implementations such as [4, 10, 45, 68] are out of scope as their use case differs from ours. The software benchmarks in [13, 16, 44] provide interesting comparisons, as the performance of the ARM Cortex-M4 is similar to that of the Ibex. However, in these works, assembly optimised SHA-256 functions are used, thus outperforming our plain Rust implementations. In addition, the authors of [13] use the rapidly verifiable approach for XMSS to lower verification time, thus outperforming the straightforward XMSS implementation of [16] by factor two. The LMS FPGA implementation in [43] supports the verification in hardware with a compact design. In comparison to our hardware/software co-design, their verification takes 20% longer in terms of absolute latency. The authors of [71] propose different hardware extensions for XMSS but conclude that any acceleration beyond the hash chain computation does not lead to improved performance. A comparison of area utilisation is not applicable, as they use a different FPGA architecture. They highly optimise their architecture for XMSS and achieve a latency which is lower by a factor of three.

As stated before, hardware/software co-designs for SPHINCS$^+$ and combined stateful and stateless HBS schemes have not been reported. The comparison with related work demonstrates that our designs improves the state-of-the-art in performance of HBSs and in their adaptability for applications.

5.4 Secure boot

In this section, we evaluate the applicability of HBSs for a post-quantum secure boot. To put our results into perspective, we include the time required to hash an application firmware image into our comparison. We use the—at time of writing—most recent Git master version[1] of the TockOS kernel as reference firmware [46] and compile it in release mode, which results into a binary size of 124KiB. To take a minimal application into account, we set the code size to 128KiB.

We compare both software only and hardware accelerated implementations. We use available open source Rust implementations for RSA[2] and ECC.[3] For comparison with hardware accelerated RSA and ECC we use the OpenTitan BigNumber accelerator (OTBN). The results of these comparisons are listed in Table 4.

In software, stateful HBSs allow for a slightly better performance for signature verification than ECC or RSA. Using a stateless HBS would mean a slight performance degradation. For implementations where the boot timing for classical asymmetric algorithms is not required to be accelerated, it is also not required for HBSs.

The hardware accelerated LMS implementation exceeds the performance of RSA and ECC run on the OTBN. In a secure boot, hashing the firmware would be the crucial part, as the verification time can be reduced to only 37% of the firmware digest time. For XMSS, the duration for signature verification doubles in comparison to RSA, but still is in a comparable range of execution time. For SPHINCS$^+$ the performance degrades for both our accelerator designs in comparison to the classical asymmetric algorithms performed on the OTBN.

To conclude, the performance of the stateful HBS schemes is comparable to that of classical asymmetric algorithms accelerated by the OTBN. It should be noted that the hardware footprint of our SHA-2$^+$ core is significantly smaller than that of the OTBN. The SPHINCS$^+$ variants degrade the performance of signature verification. However, our accelerator makes the overhead bearable. Digesting the firmware

1 https://github.com/tock/tock/commit/db7cb5f.

2 https://github.com/RustCrypto/RSA.

3 https://github.com/RustCrypto/elliptic-curves/tree/master/p256.

Table 4 Cycle count [cc] in MCycles of the signature verification executed on an Ibex processor and overhead relative to hashing a 128KiB firmware

	RSA 3072	ECC P-256	LMS w = 256	XMSS w = 16	SPX$^+$-s 256 s	SPX$^+$-r 256 s
SW	37.0	50.0	30.1	26.6	70.0	172
$cc_{verif}/cc_{sw-hash}$	141%	190%	114%	101%	266%	654%
OTBN or SHA-2$^+$	1.07	0.617	0.524	2.01	4.95	13.2
$cc_{verif}/cc_{hw-hash}$	75%	43%	37%	142%	349%	930%
SHA-2^{++}	–	–	–	–	3.26	6.61
$cc_{verif}/cc_{hw-hash}$	–	–	–	–	230%	465%

Table 5 Code size of our Rust libraries compiled for the Ibex and optimised for runtime.

	LMS	XMSS	SPX$^+$-s	SPX$^+$-r
Size in KiB	4.8	11.5	15.9 (+ 40.1)	20.8 (+ 40.1)

SPHINCS$^+$ with SHA-256 as hash function additionally requires SHA-512 to reach NIST security level 5 [38, 60]. This is due to a shortcoming in the initial SPHINCS$^+$ specification

and verification with SPX$^+$-s can be achieved below 5 MCycles.

Code size

In general, the boot ROM size is constrained. Within the OpenTitan the boot ROM has a size of 32KiB. To further allow to evaluate the applicability of HBSs, we list the required code size for our software libraries in Table 5. All HBS libraries are compiled with optimisations for runtime.

6 Implementation security

While our hardware/software co-design for HBSs enables the transition to a post-quantum secure boot, it still requires a carefully developed secure boot implementation. The secure boot concept is straightforward in theory, but its robust implementation is not. In practice, this mainly applies to memory vulnerabilities and the resilience against fault attacks. A prominent example for memory vulnerabilities is the checkm8 exploit, which affects most of Apple's devices and allows arbitrary code execution [6]. Unfortunately, this is not an isolated incident, as memory vulnerabilities are known for various devices from different manufacturers [1, 20–26, 40]. We counteract the threat of erroneous implementations regarding memory vulnerabilities by implementing our HBSs libraries in Rust. Of course, this is only one step towards an open-source bootloader written entirely in Rust, as this requires much more effort. Nevertheless, to assess the threat of fault injections to our implementations, we thoroughly analyse the fault attack resilience in the following section and discuss the need of dedicated countermeasures.

Table 6 Number of instructions, where a single-instruction-skip-fault will lead to successful verification of an invalid signature

	LMS	LMS	SPX$^+$-s	SPX$^+$-r
# of faults	6	1	3	3

6.1 Fault injection

Prominent open source secure bootloader implementations, for example MCUboot, wolfBOOT, or the OpenTitan secure boot implementation, include software-based countermeasures against fault attacks [47, 51, 72]. Some of the implementations allow to enable different levels of fault injection hardenings, each of which increases the resilience but consequently also the overhead in terms of performance and code size. Within this work, we define the protection against a fault model of a single instruction skip as minimum requirement. Such a fault model may allow an adversary to boot an image with an invalid signature and is also used to evaluate the resilience of the bootloaders listed above. We employ the open source ARCHIE framework [32] to find vulnerabilities to such a fault model in our code. Our tests cover the three implementations of LMS, XMSS, and SPHINCS$^+$. During the tests we apply the instruction skip fault model to each executed instruction to test whether an invalid signature image pair is assessed as valid.

Table 6 lists the results of our experiments. For LMS, two unique locations were identified, which allow to tamper with the signature verification routine. The first vulnerable location corresponds to an instruction that can be faulted at five different points in time over the complete execution of the verification routine. It is an AUIPC (*add upper immediate to pc*) instruction, which is part of a for-loop using the authentication data to calculate the top-node of the Merkle tree, i.e. the LMS public key candidate. For this analysis, we use a signature containing a Merkle tree with five layers. In each for-loop iteration, this instruction is executed and therefore could possibly be tampered to manipulate the signature verification. As the instruction adds an immediate value to the program counter and stores it in a destination register, it is used to calculate addresses relative to the program counter to prepare a branch. If the instruction is skipped, the verification routine prematurely returns an *Ok* value to the calling function, which is wrongly assumed as successful signature verification. This is due to the fact that the authentication path calculation returns the public key candidate, encapsulated with a *Result*, that—in case of a fault injection—is passed directly to the verification routine. A simple re-factoring, such that the authentication path calculation only returns the public key candidate, and the *Result* value *Ok* is returned independently by the sub-function calling the authentication path calculation should be enough to mitigate this vulnerability. In the event of a fault attack, the verification routine will detect the tampering, as it expects a *Result* and gets a plain data array. The second vulnerable location is a branch instruction, which stems from a *map_err* routine. This routine is called to handle the returned value if *Result* is set to *Error*, so obviously skipping the check means that an invalid signature is verified successfully. A simple countermeasure consists of a duplicated check and ensuring that the compiler does not remove the—seemingly unnecessary—second operation.

For XMSS, a single vulnerable location was identified, which allows to tamper with the signature verification routine. This is also related to the check of the return value of the signature verification routine and can consequently be eliminated by duplicating the check.

For the two variants of SPHINCS$^+$, robust and simple, three vulnerable locations were identified, which allow to tamper with the signature verification routine. The first is a conditional set instruction within the *compare* routine for the *Slice* type. This routine is used to compare the public key with its candidate. Again, duplication is sufficient to mitigate the threat. The second vulnerability is a jump instruction to enter the signature verification routine. This call has to be hardened, such that it is detected if the routine is skipped. Such a countermeasure can be designed by adding so-called prologs and epilogs to function calls (see next paragraph). The third location is a branch instruction that stems from the *assert_eq!* macro to check the returned value in the main routine. As before, this can be easily hardened by duplicating the statement.

Realising software-based countermeasures

The findings of the ARCHIE experiments can be grouped into the two categories of *I)* calling a function, and *II)* handling of data. In the case of *I)* calling a function, a countermeasure may be designed similar to the prolog and epilog approach of the MCUboot fault injection hardenings [51]. Prolog and epilog correspond to code that is automatically inserted before and after each call that ensures that the function was called and returned as expected. Due to the differences of Rust and C, future work is required to design such a countermeasure in Rust. For *II)* handling data, we demonstrated that duplication is often sufficient to mitigate the threat of a single instruction skip. However, for the duplication countermeasure to be effective, any compiler optimisation to erase the seemingly unnecessary duplicated operations must be evaded. One promising approach may be to cast the values into a volatile type. Subsequently, the *core::ptr::read_volatile* function must be used to access the values. This forbids the compiler to detect the duplicated statement and as a result it does not remove it. The drawback of this approach is that it requires an *unsafe* statement, which should be avoided in Rust if possible. Further, we want to emphasise that, while we see our analysis as a reasonable approach to apply software-based countermeasures and verify their effectiveness, we do not deem it an ideal solution. Ideally there would be a dedicated crate, which offers fault injection hardened replacements of frequently used methods regarding calling functions, returning values, or *assert* macros. The *subtle* crate demonstrates the realisation of such a concept for constant-time cryptographic implementations. To the best of our knowledge there are currently no public crates available which cover fault injection hardenings.

Limitations of the instruction-skip fault model

So far, the resilience analysis in this section was limited to a single-instruction-skip fault model. In contemporary literature on fault attacks, this is a common practice for investigating attacks and designing countermeasures [7, 8, 56]. In reality, however, this fault model is a vast simplification of the physical effects of fault injection and obviously limited to software implementations. For software implementations, the fault model can be extended to cover arbitrary instruction-skips and faults that tamper data values in ALU registers. Research that focuses on the safety domain of embedded systems in hazardous environments covers all possible effects that are introduced by the environment (e.g. radiation) with two major groups: data-flow and control flow corruption [29, 30, 57, 58, 61, 66, 70]. Obviously, if arbitrary control and data-flow corruption is considered, software-based countermeasures can no longer simply be realised with a few changes to high-level (Rust or C) source code. Instead, to encounter data-flow errors, the ALU registers are divided into a primary and a shadow set, that "shadows" the values of the primary registers. Every assembly instruction is duplicated, such that the operands of the duplicated instructions are the shadow registers of the primary registers [29, 30, 58, 61]. Control-flow corruption is mitigated by inserting signature checks between blocks of assembly instructions [57, 70]. In [65], an integration of these concepts into the clang toolchain was presented, such that code is hardened automatically during compilation. Obviously, the overhead of these countermeasures far exceeds the minimal overhead needed to mitigate the threat of single-instruction skips. In [33], steps to transfer these concepts from the safety domain, where faults comply with a common noise model, to the security domain, where faults are more precise, were taken. For this work, we deem the protection against the single-instruction-skip fault model sufficient.

Hardware-based countermeasures

Generic hardware-based countermeasures are mostly built around redundancy, integrity codes, and encodings with high hamming distance, such that getting from one valid encoding to another by means of fault injection is hard. The OpenTitan [47] incorporates such countermeasures for its memory, registers, bus interface, finite state machines (FSMs) of peripherals, and even has a complete lockstep core, that duplicates important logic of the CPU to detect inconsistent results between the two cores.

The protection of cryptographic circuits requires more sophisticated approaches. Cryptographic algorithm are subject to the threat of differential fault analysis (DFA) [12] and related attacks, that leak secret information to an adversary. The complete duplication of a cryptographic circuit to detect faults is possible, but with precise fault injection methods, an adversary could—in theory—inject the same error into two parts of the chip. In [2, 64], an approach to combine redundancy with encoded data that is processed concurrently to the original data, was proposed. In [63], this theory was applied to AES. Another interesting approach based on gadgets that provide both resilience against (passive) side-channel attacks and (active) fault attacks was proposed in [27, 28].

Recently, the verification of fault injection countermeasures gained more interest. Frameworks such as [54, 62] allow to verify a netlist regarding active security, by inserting up to k faults into the circuit model, e.g. by inverting wires.

As stated above, we deem the protection against single-instruction-skips sufficient. To boot an image with an invalid signature, advanced attacks, such as differential fault analysis (DFA), on signature verification and the underlying hash function of the HBS scheme are of no help. Therefore, an adversary is limited to faults that force the verification routine to accept the invalid signature. Targeting the hash core does not make sense in such a scenario. Further, our hardware design is hardened with generic OpenTitan countermeasures, as all its peripherals. For signature generation in HBS multi-tree schemes, the integration of approaches such as [2, 28] makes sense, as these schemes are extremely vulnerable to fault attacks [3, 17]. In this case, the software also must be hardened more thoroughly, e.g. with the approaches introduced in the previous paragraph. We leave such a design open for future work.

7 Conclusion and outlook

In this work, we show that the transition to a post-quantum secure boot using schemes is feasible for today's schemes is feasible for today's designs. In contrast to other works, we provide a flexible hardware/software co-design to support both stateful as well as stateless schemes from boot up. We demonstrate that by exploiting similarities of LMS or XMSS and SPHINCS$^+$ low hardware overheads can be achieved. Hence, making the discussion to choose between stateless and stateful HBSs one indifferent to the underlying hardware. Further, our design allows to easily incorporate updates with respect to the parameters without changes to the hardware design.

Regarding the parameter sets that should be chosen for secure boot with HBSs, we come to the following conclusions: As NIST views both "simple" and "robust" constructs as secure [19] we recommend the usage of LMS and SPHINCS$^+$-s, due to their advantageous performance. Our design demonstrates that both can be implemented with a minimal hardware footprint. The synergy between LMS and SPHINCS$^+$-s makes them ideal for a flexible architecture. Although, it should be noted that the small differences between the OTS in LMS and SPHINCS$^+$-s make the design more complicated than it needs to be. For XMSS and SPHINCS$^+$-r this is even more obvious. From an implementers perspective, a uniform approach for all "simple" and all "robust" constructs would be desirable constructs would be desirable. We established that our architecture allows to choose $w = 256$ for LMS without a significant performance penalty. Due to the small signature size, this is our Winternitz parameter of choice.

We would like to highlight that our design is suited for fast transition, as well as being a starting point for further research, in particular for designs and parameter explorations of SPHINCS$^+$. NIST even requested public feedback for new SPHINCS$^+$ parameter sets [53]. While their focus is a lower number of maximum signatures, we suggest to revisit higher Winternitz parameters. As shown in this work, on hardware/software co-designs this does not degrade the performance but reduces the signature size. With this paper we put forward that hardware/software co-designs are the most relevant solution for problems like secure boot. Software only implementations will not be suitable for embedded devices due to timing requirements and full hardware implementations are very expensive in terms of overhead and cost. Dedicated hardware/software co-designs for SPHINCS$^+$ are largely unexplored. With our flexible HBS design for secure boot we hope to motivate more research in this direction.

Acknowledgements This work was partly funded by the German Ministry of Education, Research and Technology in the context of the project Aquorypt (Reference Number 16KIS1018). We thank Chris Gourley, Yongkui Han, Steve Rich, Chris Shenefiel, and Chirag Shroff from CISCO Systems, Inc. for the valuable discussions and feedback, especially on LMS. The research work on the extended version was supported by the Bavarian Ministry of Economic Affairs, Regional Development and Energy, in the project "Trusted Electronics Center Bavaria."

Author Contributions All authors contributed equally to this work.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Abbott, L.: Another vulnerability in the LPC55S69 ROM. https://oxide.computer/blog/another-vulnerability-in-the-lpc55s69-rom. March (2022)
2. Aghaie, A., Moradi, A., Rasoolzadeh, S., Shahmirzadi, A.R., Schellenberg, F., Schneider, T.: Impeccable circuits. Cryptology ePrint Archive, Paper 2018/203, (2018) https://eprint.iacr.org/2018/203
3. Amiet, D., Leuenberger, L., Curiger, A., Zbinden, P.: Fpga-based sphincs+ implementations: mind the glitch. In: 2020 23rd Euromicro Conference on Digital System Design (DSD), pp. 229–237 (2020)
4. Amiet, D., Leuenberger, L., Curiger, A., Zbinden, P.: Fpga-based sphincs+ implementations: mind the glitch. In: 2020 23rd Euromicro Conference on Digital System Design (DSD), pp. 229–237, Kranj, Slovenia. IEEE (2020)
5. ANSSI. ANSSI views on the post-quantum cryptography transition, January (2022)
6. Apple devices vulnerable to arbitrary code execution in SecureROM. https://www.kb.cert.org/vuls/id/941987/. December (2019)
7. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proc. IEEE **94**(2), 370–382 (2006)
8. Barry, T., Couroussé, D., Robisson, B.: Compilation of a countermeasure against instruction-skip fault attacks. In: Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, pp. 1–6 (2016)
9. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'sHearn, Z.: SPHINCS: practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.), Advances in Cryptology—EUROCRYPT 2015, Lecture Notes in Computer Science, vol. 9056, pp. 368–397. Springer, Berlin, January (2015)
10. Berthet, Q., Upegui, A., Gantel, L., Duc, A., Traverso, G.: An area-efficient SPHINCS+ post-quantum signature coprocessor. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 180–187, Portland, OR, USA. IEEE (2021)
11. Beullens, W.: Breaking rainbow takes a weekend on a laptop (2022)
12. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO'97, pp. 513–525. Springer, Berlin (1997)
13. Bos, J.W., Hülsing, A., Renes, J., van Vredendaal, C.: Rapidly verifiable XMSS signatures. In: IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(1), pp. 137–168 (2020)
14. BSI. Bsi - technische richtlinie: Kryptographische verfahren: Empfehlungen und schluessellaengen, January (2022)
15. Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. In: Katz, J., Yung, M. (eds.) Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 4521, pp. 31–45. Springer, Berlin (2007)
16. Campos, F., Kohlstadt, T., Reith, S., Stöttinger, M.: LMS vs XMSS: comparison of stateful hash-based signature schemes on ARM cortex-m4. In: Nitaj, A., Youssef, A. (eds.) Progress in Cryptology - AFRICACRYPT 2020. Lecture Notes in Computer Science, vol. 12174, pp. 258–277. Springer, Cham (2020)
17. Castelnovi, L., Martinelli, A., Prest, T.: Grafting trees: a fault attack against the sphincs framework. Cryptology ePrint Archive, Paper 2018/102 (2018). https://eprint.iacr.org/2018/102
18. CISCO, April (2019)
19. Cooper, D.A., Apon, D.C., Dang, Q.H., Davidson, M.S., Dworkin, M.J., Miller, C.A.: Recommendation for stateful hash-based signature schemes (2020)
20. CVE-2017-7932. https://nvd.nist.gov/vuln/detail/CVE-2017-7932 (2017)
21. CVE-2017-7936. https://nvd.nist.gov/vuln/detail/CVE-2017-7936 (2017)
22. CVE-2018-6242. https://nvd.nist.gov/vuln/detail/CVE-2018-6242 (2018)
23. CVE-2021-0467. https://nvd.nist.gov/vuln/detail/CVE-2021-0467 (2021)
24. CVE-2021-40154. https://nvd.nist.gov/vuln/detail/CVE-2021-40154 (2021)
25. CVE-2021-44850. https://nvd.nist.gov/vuln/detail/CVE-2021-44850 (2021)
26. CVE-2022-22819. https://nvd.nist.gov/vuln/detail/CVE-2022-22819 (2022)
27. Dhooghe, S., Nikova, S.: Let's tessellate: tiling for security against advanced probe and fault adversaries. Cryptology ePrint Archive, Paper 2020/1146 (2020). https://eprint.iacr.org/2020/1146
28. Dhooghe, S., Nikova, S.: My gadget just cares for me—how Nina can prove security against combined attacks. Cryptology ePrint Archive, Paper 2019/615 (2019). https://eprint.iacr.org/2019/615
29. Didehban, M., Shrivastava, A., Lokam, S.R.D.: Nemesis: a software approach for computing in presence of soft errors. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 297–304 (2017)
30. Didehban, M., Shrivastava, A.: NZDC: a compiler technique for near zero silent data corruption. In: 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2016)
31. Dods, C., Smart, N.P., Stam, M.: Hash based digital signature schemes. In: Smart, N.P. (ed.) Cryptography and Coding, pp. 96–115. Springer, Berlin (2005)
32. Fraunhofer AISEC. Archie. September (2021)
33. Geier, J., Auer, L., Mueller-Gritschneder, D., Sharif, U., Schlichtmann, U.: Compasec: a compiler-assisted security countermeasure to address instruction skip fault attacks on RISC-V. In: Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC'23, pp. 676–682. Association for Computing Machinery, New York (2023)
34. Ghosh, S., Misoczki, R., Sastry, M.R.: Lightweight post-quantum-secure digital signature approach for IoT motes (2019)

35. Groot Bruinderink, L., Hülsing, A.: “oops, i did it again”-security of one-time signatures under two-message attacks. In: Adams, C., Camenisch, J. (eds.) Selected Areas in Cryptography - SAC 2017. Lecture Notes in Computer Science, vol. 10719, pp. 299–322. Springer, Cham (2018)
36. Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. https://datatracker.ietf.org/doc/html/rfc8391. May (2018)
37. Hülsing, A., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.-L., Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Aumasson, J.-P., Westerbaan, B., Beullens, W.: SPHINCS+ - submission to the NIST post-quantum project, v.3.1. https://sphincs.org/data/sphincs+-r3.1-specification.pdf (2022)
38. Hülsing, A., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.-L., Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Aumasson, J.-P., Westerbaan, B., Beullens, W.: SPHINCS+ round 3 presentation. https://csrc.nist.gov/CSRC/media/Presentations/sphincs-round-3-presentation/images-media/session-1-sphincs-plus-hulsing.pdf. June (2021)
39. Hülsing, A., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.-L, Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Aumasson, J.-P., Westerbaan, B., Beullens, W.: SPHINCS+ - submission to the NIST post-quantum project, 3. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. (2020)
40. Hyvärinen, N.: Break your own product, and break it hard. https://blog.f-secure.com/break-your-own-product-and-break-it-hard/. July (2017)
41. IHP. SiGe-BiCMOS- und Siliziumphotonik-Technologien
42. Kampanakis, P., Fluhrer, S.: LMS vs XMSS: Comparion of two hash-based signature standards (2017)
43. Kampanakis, P., Panburana, P., Curcio, M., Shroff, C.: Post-quantum hash-based signatures forÂ secure boot. In: Park, Y., Jadav, D., Austin, T. (eds.) Silicon Valley Cybersecurity Conference, pp. 71–86. Springer, Cham (2021)
44. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: post-quantum crypto library for the ARM Cortex-M4 (2018). https://github.com/mupq/pqm4
45. Kumar, V.B.Y., Gupta, N., Chattopadhyay, A., Kasper, M., Krauß, C., Niederhagen, R.: Post-quantum secure boot. In: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1582–1585, Grenoble, France, 2020. IEEE
46. Levy, A., Campbell, B., Ghena, B., Giffin, D.B., Pannuto, P., Dutta, P., Levis, P.: Multiprogramming a 64kb computer safely and efficiently. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP’17, 10, pp. 234–251. ACM, New York (2017)
47. lowRISC. Opentitan documentation (2022)
48. MATZOV. Report on the Security of LWE: Improved Dual Lattice Attack, April (2022)
49. McGrew, D., Fluhrer, S., Curcio, M.: Leighton-Micali Hash-Based Signatures. https://datatracker.ietf.org/doc/html/rfc8554. April (2019)
50. McGrew, D., Kampanakis, P., Fluhrer, S., Gazdag, S.-L., Butin, D., Buchmann, J.: State management for hash-based signatures. In: Chen, L., McGrew, D., Mitchell, C. (eds.) Security Standardisation Research. Lecture Notes in Computer Science, vol. 10074, pp. 244–260. Springer, Cham (2016)
51. MCUboot. Mcuboot documentation (2022)
52. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology-CRYPTO’89 Proceedings. Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer, New York (1990)
53. Moody, D., Alagic, G., Apon, D.C., Cooper, D.A., Dang, Q.H., Kelsey, J.M., Liu, Y.-K., Miller, C.A., Peralta, R.C., Perlner, R.A., Robinson, A.Y., Smith-Tone, D.C., Alperin-Sheriff, J.: Status report on the third round of the NIST post-quantum cryptography standardization process (2022)
54. Nasahl, P., Osorio, M., Vogel, P., Schaffner, M., Trippel, T., Rizzo, D., Mangard, S.: Synfi: pre-silicon fault analysis of an open-source secure element. IACR Trans. Cryptogr. Hardw. Embedded Syst. **2022**(4), 56–87 (2022)
55. NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2016). https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf
56. O’Flynn, C.: MIN()imum failure: EMFI attacks against USB stacks. In: 13th USENIX Workshop on Offensive Technologies (WOOT 19), Santa Clara, CA, August. USENIX Association (2019)
57. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-flow checking by software signatures. IEEE Trans. Reliab. **51**(1), 111–122 (2002)
58. Oh, N., Shirvani, P.P., McCluskey, E.J.: Error detection by duplicated instructions in super-scalar processors. IEEE Trans. Reliab. **51**(1), 63–75 (2002)
59. Perin, L.P., Zambonin, G., Martins, D.M.B., Custódio, R., Martina, J.E.: Tuning the winternitz hash-based digital signature scheme. In: 2018 IEEE Symposium on Computers and Communications (ISCC), Natal, Brazil, pp. 00537–00542. IEEE (2018)
60. Perlner, R., Kelsey, J., Cooper, D.: Breaking Category Five SPHINCS+ with sha-256. August (2022)
61. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: Swift: software implemented fault tolerance. In: International Symposium on Code Generation and Optimization, pp. 243–254. IEEE (2005)
62. Richter-Brockmann, J., Feldtkeller, J., Sasdrich, P., Güneysu, T.: Verica—verification of combined attacks: automated formal verification of security against simultaneous information leakage and tampering. Cryptology ePrint Archive, Paper 2022/484 (2022). https://eprint.iacr.org/2022/484
63. Richter-Brockmann, J., Sasdrich, P., Bache, F., Güneysu, T.: Concurrent error detection revisited: Hardware protection against fault and side-channel attacks. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, ARES’20. Association for Computing Machinery, New York (2020)
64. Shahmirzadi, A.R., Rasoolzadeh, S., Moradi, A.: Impeccable circuits ii. Cryptology ePrint Archive, Paper 2019/1369 (2019). https://eprint.iacr.org/2019/1369
65. Sharif, U., Mueller-Gritschneder, D., Schlichtmann, U.: Compas: compiler-assisted software-implemented hardware fault tolerance for RISC-V. In: 2022 11th Mediterranean Conference on Embedded Computing (MECO), pp. 1–4 (2022)
66. Sharif, U., Mueller-Gritschneder, D., Schlichtmann, U.: Repair: Control flow protection based on register pairing updates for SW-implemented hw fault tolerance. ACM Trans. Embedded Comput. Syst. (TECS) **20**(5s), 1–22 (2021)
67. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer (1997)
68. Thoma, J.P., Guneysu, T.: A configurable hardware implementation of XMSS (2021)
69. United States government National Security Agency. Commercial national security algorithm suite 2.0 (2022)
70. Vankeirsbilck, J., Penneman, N., Hallez, H., Boydens, J.: Random additive signature monitoring for control flow error detection. IEEE Trans. Reliab. **66**(4), 1178–1192 (2017)
71. Wang, W., Jungk, B., Wälde, J., Deng, S., Gupta, N., Szefer, J., Niederhagen, R.: Xmss and embedded systems. In: Paterson, K.G.,

Stebila, D. (eds.) Selected Areas in Cryptography – SAC 2019, pp. 523–550. Springer, Cham (2020)
72. wolfBOOT. wolfboot documentation (2023)