Faulting Winternitz One-Time Signatures to forge LMS, XMSS, or SPHINCS^{+} signatures

Alexander Wagner[1,2][0000-0003-2853-3063],
Vera Wesselkamp[2][0009-0002-8481-9103], Felix Oberhansl[1][0000-0002-7822-2880],
Marc Schink[1,2], and Emanuele Strieder[1,2][0000-0003-0204-3234]

[1] Fraunhofer Institute for Applied and Integrated Security (AISEC),
Garching near Munich, Germany
firstname.surname@aisec.fraunhofer.de
[2] Technical University of Munich,
Munich, Germany
firstname.surname@tum.de

Keywords: fault injection · post-quantum cryptography · hash-based signatures · winternitz one-time signatures · LMS · XMSS · SPHINCS+

Abstract. Hash-based signature (HBS) schemes are an efficient method of guaranteeing the authenticity of data in a post-quantum world. The stateful schemes LMS and XMSS and the stateless scheme SPHINCS^{+} are already standardised or will be in the near future. The Winternitz one-time signature (WOTS) scheme is one of the fundamental building blocks used in all these HBS standardisation proposals. We present a new fault injection attack targeting WOTS that allows an adversary to forge signatures for arbitrary messages. The attack affects both the signing and verification processes of all current stateful and stateless schemes. Our attack renders the checksum calculation within WOTS useless. A successful fault injection allows at least an existential forgery attack and, in more advanced settings, a universal forgery attack. While checksum computation is clearly a critical point in WOTS, and thus in any of the relevant HBS schemes, its resilience against a fault attack has never been considered. To fill this gap, we theoretically explain the attack, estimate its practicability, and derive the brute-force complexity to achieve signature forgery for a variety of parameter sets. We analyse the reference implementations of LMS, XMSS and SPHINCS^{+} and pinpoint the vulnerable points. To harden these implementations, we propose countermeasures and evaluate their effectiveness and efficiency. Our work shows that exposed devices running signature generation or verification with any of these three schemes must have countermeasures in place.

1 Introduction

Hash-based signature (HBS) schemes have been known for decades but they were not really considered for further research or practical applications in the past.

This changed when the need for post-quantum cryptography (PQC) emerged that could withstand attacks by quantum computers.

The standardization of stateful HBS schemes started with the publications of the IETF RFCs for the eXtended Merkle Signature Scheme (XMSS) and Leighton-Micali hash-based signature (LMS) in 2018 and 2019, respectively [HBG$^+$18, MMC19]. The National Institute of Standards and Technology (NIST) published a supplement to their digital signature standard recommending parameters for both of these algorithms in 2020 [CAD$^+$]. The French national agency for the security of information systems (ANSSI) and the German federal office for information security (BSI) also specify both algorithms in their own publications [ANS22, BSI22]. The stateless scheme SPHINCS$^+$ was selected in 2022 at the end of the third round of the process to standardize quantum-resistant public key cryptographic algorithms [MAA$^+$].

Since their standardization, stateful HBS algorithms have been deployed in several products ranging from embedded devices up to servers [Rai22, Cis19, gen20]. Due to their inherent nature of statefulness, the number of signatures that can be created with a key pair is limited, which also limits the range of applications. In practice, they are most applicable to verify the integrity and authenticity of data that rarely changes, such as the firmware of embedded devices. The verification procedure then takes place during a secure boot or firmware update process. In past works, the research community has investigated hardware and software optimizations for this use case [WOS22, WJW$^+$, KPC$^+$, KGC$^+$20] and vendors brought forward products [Rai22]. The standard for SPHINCS$^+$ is yet to be published sometime between today and 2024, but the scheme is already considered for adoption [MAA$^+$]. For example, the OpenTitan project considers to integrate SPHINCS$^+$ into their open source hardware root of trust for firmware verification [goo].

These efforts demonstrate the need for a post-quantum secure boot and firmware update process. An adversary who can circumvent such a process can execute malicious firmware, which compromises the security of embedded devices completely. Over time, researchers have established that fault attacks pose a considerable threat to exposed embedded devices, e.g. by allowing exactly such a circumvention of the secure boot process [BFP19, Rot19]. Developers of secure boot libraries such as MCUboot[3] and microcontroller manufacturers have recognized this by introducing countermeasures against such attacks in the basic control flow [AdGHB]. The cryptographic implementations, however, remained unprotected. We present a fault attack, which demonstrates that such an assumption could prove fatal for an exposed embedded device that uses any of the three HBS schemes: LMS, XMSS, and SPHINCS$^+$.

Attack Overview. Instead of trying to entirely skip a secure boot or firmware update process, our fault attack targets the internal structure of HBS schemes. Our attack grants the adversary signature forgery for arbitrary malicious payloads. We want to emphasize the impact of such an attack, if executed successfully. It is

[3] `https://github.com/mcu-tools/mcuboot`

common practice to rely on one entity for signing firmware updates with one key pair for a complete line of products. Therefore, forging a single signature for a malicious payload that seems valid with respect to the entity's public key allows the adversary to corrupt any device. We introduce the idea behind the attack itself in detail in Section 3 and the adversarial model in Section 3.5. The attack can target either the signing or the verifying entity, is applicable to LMS, XMSS, and SPHINCS$^+$, and consists of two phases. In one phase, a fault is introduced into the Winternitz one-time signature (WOTS) signing or verifying procedure. The other phase is responsible for brute-forcing a forgery candidate. The order of these phases depends on whether it is applied to the signer or the verifier. Further, the effort and success probability of the brute-force phase (analyzed in Section 4) depends on the algorithmic part targeted during fault injection. We demonstrate how our attack can be used on the reference implementations of LMS, XMSS, and SPHINCS$^+$ in Section 6.

Related Work. The only fault attack known in the context of HBS is the 'Grafting Trees' attack, proposed in [CMP18]. Effective and efficient countermeasures have not yet been sufficiently developed [Gen23]. Practicable evaluations of this fault injection attack were shown in [GKPM18, ALCZ20]. It targets the signature generation of multi-tree schemes, therefore it only affects SPHINCS$^+$ and the multi-tree variants of LMS and XMSS. The adversary tampers with the signing procedure, such that the signer unknowingly leaks secret information. A few tries suffice for the adversary to be able to reconstruct the signer's secret key. The attack has the advantage of very lax requirements with respect to the fault model and the temporal precision. Once the adversary extracts the secret key, she can sign arbitrary messages. The disadvantage is that the attack can only be carried out on the signer. Therefore, it is not applicable in the context of firmware updates or secure boot, as the adversary typically does not have physical access to the signing entity.

Contributions. We present the first attack that allows to tamper the signing as well as the verification operation of HBS schemes in general. Our attack is applicable to all variants of LMS, XMSS, and SPHINCS$^+$, by targeting the checksum mechanism in the fundamental WOTS scheme. The attack consists of two phases. The first phase is a brute-force search for a suitable message digest. This phase happens offline, i.e. there are no strict timing requirements on the adversary. We derive the brute-force complexity and success probability depending on the fault model and algorithmic parameters. Further, we estimate the cost of the brute-forcing capabilities needed in practice. The second phase covers the physical attack on the victim device, typically an embedded system to which the adversary has physical access. We analyze real world implementations for weak spots and show the applicability of our attack with respect to the capabilities of the adversary. In combination with the brute-force cost, our analysis shows that attacking reference implementations of all considered HBS schemes, namely LMS, XMSS, and SPHINCS$^+$ is feasible. To conclude, we outline different countermeasures to mitigate our attack, estimate their effectiveness and costs, and stress their importance for exposed devices.

2 Hash-Based Signatures

We briefly introduce the structure of WOTS and explain how it is used as a fundamental building block in the HBS algorithms LMS [MMC19], XMSS [HBG⁺18], and SPHINCS$^+$ [HBD$^+$20].

2.1 Winternitz One-Time Signatures

Figure 1 depicts the principle behind WOTS+ as described in [Hü]. In the following we refer to WOTS+ as WOTS, unless clearly stated otherwise, as it is the foundation for all "WOTS-like" algorithms in LMS, XMSS, and SPHINCS$^+$, the most relevant HBS schemes to date.

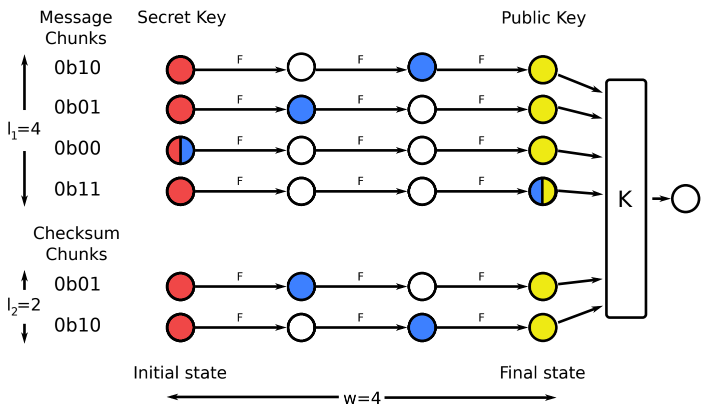


Fig. 1: Simplified Winternitz one-time signature – $w = 4$ and $n = 1$ – with the nodes of the secret key (🔴), the public key (🟡), and the signature (🔵) highlighted.

A WOTS signature consists of l hash chains and of these l_1 are required for the message digest and l_2 for the checksum, which are defined as

$$l = l_1 + l_2,\ l_1 = \left\lceil \frac{8n}{log_2(w)} \right\rceil,\ l_2 = \left\lfloor \frac{log_2(l_1(w-1))}{log_2(w)} \right\rfloor + 1. \tag{1}$$

To generate a WOTS signature, a message is hashed into an n-byte value m. The message digest m is split into l_1 chunks. Each chunk is interpreted as a value $m_i = \mathcal{N}(m, i)$, i.e. the function $\mathcal{N}$ maps the i-th chunk of m to m_i, where $m_i \in [0, w-1]$ and $i \in [0, l_1 - 1]$. The parameter w is the Winternitz parameter. Each of the values m_i is assigned an individual hash chain consisting of w nodes, each represented by an n-byte value. The start node is the one-time signature

(OTS) secret key (🔴), and the end node the OTS public key (🟡). Advancing from one node to another is realized by applying a function $\mathcal{F}$ to the current node. The output of $\mathcal{F}$ serves as the next node. The end nodes are combined by applying the function $\mathcal{K}$ to obtain the compressed OTS public key. Although the exact implementations of $\mathcal{F}$ and $\mathcal{K}$ may differ, we assume both to be single calls to a cryptographic hash function. In reality, before being hashed, the node data might be - depending on the scheme - pre-processed with masks and keys, which are also the output of a hash function.

To **sign** (🔴→🔵) or **verify** (🔵→🟡) a m_i the corresponding hash chain is advanced by applying $\mathcal{F}$. For signing, $\mathcal{F}$ is applied m_i times to the respective secret key node (🔴) and the resulting node (🔵) is taken as part of the WOTS signature. For verifying, the signature node (🔵) is taken as basis and advanced $w - 1 - m_i$ times. If this does not yield the public key (🟡), the verifier rejects the signature.

If the WOTS scheme were used just with the l_1 hash chains representing the message digest m, an adversary could trivially sign any message, where the digest r consists only of chunks r_i, where $r_i \geq m_i, \forall i \in [0, l_1 - 1]$. This is because the adversary gains information about intermediate hash chain nodes from the original signature. Information that was prior to the signing operation, private. The adversary can simply advance all signature nodes by $r_i - m_i$ to forge a signature. To mitigate this, a checksum mechanism is part of the WOTS scheme. In addition to the message digest and its corresponding signature nodes, each WOTS signature consist also of a checksum c, which has its own signature nodes (Figure 1). The calculation of the checksum c for a message digest m is denoted as

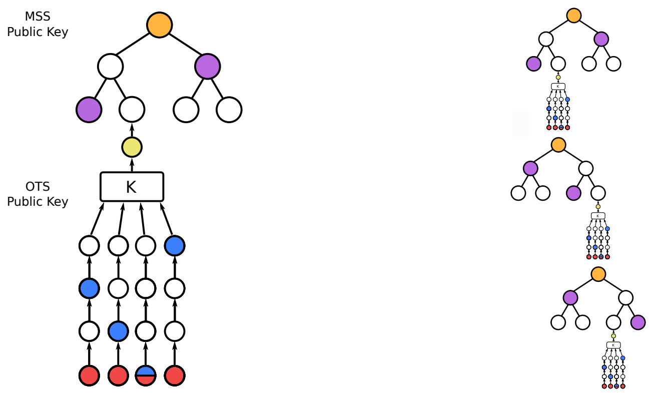
$$c = \mathcal{C}(m) = \sum_{i=0}^{l_1-1} (w - 1 - m_i). \tag{2}$$

Put in simple terms, c corresponds to the sum of "steps left" over all message hash chains. The value c is split into l_2 checksum chunks c_k, where $k \in [0, l_2 - 1]$ and l_2 is defined in Equation (1). The mapping between c and checksum chunks c_k is defined by the function $\mathcal{N}(c, k)$, similar to the mapping between message digest m and message chunks m_i. For the final signature, message chunks and checksum chunks are appended, s.t. $m_0 \,|\, m_1 \,|\, \ldots \,|\, m_{l_1-1} \,|\, c_0 \,|\, c_1 \,|\, \ldots \,|\, c_{l_2-1}$. By doing an index transformation from $k \in [0, l_2 - 1]$ to $j \in [l_1, l - 1]$, we map $c_k = m_j$, s.t. we can simplify our signature to a continuous series of $m_0 \,|\, m_1 \,|\, \ldots \,|\, m_{l-1}$, where m_i are nodes corresponding to message chunks and m_j are nodes corresponding to checksum chunks. With the checksum nodes, it is now guaranteed that for a malicious message digest r for which $r_i \geq m_i, \forall i \in [0, l_1 - 1]$ the checksum $c' < c$. Therefore, the adversary would have to get to a lower node from a higher node for at least one checksum chain. This is impossible from an algorithmic perspective, as these lower nodes are neither public nor computable.

The WOTS scheme used today (WOTS+ [Hü]) is a result of optimizing the original scheme by Winternitz [Mer90] and the updated version from [BDE$^+$11]. Its actual instantiations in LMS, XMSS and SPHINCS$^+$ differ, but the parts relevant for this paper are equivalent. This includes the Winternitz one-time signature with tweakable hash functions (WOTS-TW) scheme, the WOTS scheme used in SPHINCS$^+$, which was formally extracted and equipped with a new security proof in [HK22], after a flaw in the original proof was found. The Compressed Winternitz one-time signature (WOTS+C) scheme [KHRY22], however, differs in the fact that no checksum chains are required. Instead, a short random bit string (salt) is introduced in the signing procedure. The salt is sampled randomly until a message digest with a pre-defined value for c is found. This modification was proposed as part of the efforts to compress SPHINCS$^+$ signatures, as it makes the checksum signature nodes obsolete.

2.2 LMS, XMSS, and SPHINCS$^+$

For most of today's applications of digital signatures, a one-time signature scheme like WOTS alone can hardly ever be used. Therefore, many-time signatures (MTSs) like XMSS and LMS combine WOTS with one or multiple Merkle trees. The idea of Merkle signature schemes (MSSs) can be traced back to [Mer90]. Its structure is depicted in Figure 2a. These schemes are stateful, i.e. the amount of signatures that can be created with one key pair is greater than one but still limited and the signer needs to keep track of the signatures that were already used (maintain a state). As in the previous subsection, WOTS is used to sign the initial message digest. The WOTS public key nodes (●) correspond to the leaf nodes of a Merkle tree. The root node of the tree (●), in turn, corresponds to the LMS or XMSS public key. Therefore, a Merkle tree with a tree height of h can authenticate 2^h WOTS key pairs, each of which can be used once.



(a) The MSS scheme. (b) The GMSS scheme.

Fig. 2: MTS variants with one (MSS) and multiple (GMSS) levels of Merkle trees.

To sign a message, the signing entity publishes the WOTS signature (●) and the so-called authentication path (●). These nodes are used by the verifying entity to compute the root node and check whether it matches the MSS public key (●).

For a large number of signatures, MSS schemes still proof impractical. For larger tree heights h, the runtime of key and signature generation is no longer feasible. The generalized Merkle signature scheme (GMSS) [BDK$^+$] addresses this issue, by stacking up d Merkle trees of smaller height $h' = h/d$ instead of using one large tree of height h (see Figure 2b). The WOTSs of the top and intermediate Merkle trees are used to authenticate the root nodes of the respective Merkle trees below (sub-trees). The WOTSs of the Merkle trees on the lowest layer are used to sign message digests. The multi-tree variants of XMSS, XMSS-MT, and LMS, hierarchical signature system (HSS), specify different parameter sets for d and h, which can be chosen depending on the number of required signatures.

For SPHINCS$^+$, a limited subset of parameters exist. The SPHINCS$^+$ scheme is an stateless signature scheme. Theoretically, a bound for the maximum number of allowed signatures can be derived, but due to a careful combination of parameters, this number is too high to pose a limitation for real-world applications. Additionally, the message digest in SPHINCS$^+$ is not signed with WOTS, but a few-time signature (FTS) scheme called forest of random subsets (FORS). Since the usage of FORS is not of importance for our attack, we omit an explanation here and refer the interested reader to [HBD$^+$20] instead.

3 Attack Sketch

Our attack enables an adversary to choose an arbitrary message and create a valid signature, which we refer to as forged signature throughout this paper. The forged signature can be generated if the adversary has at least one signature which was signed with the secret key. In contrast to existing fault attacks in the context of HBSs, the adversary can either target the signing or the verifying entity. In the following, we abbreviate the two scenarios with $\mathcal{FS}$ and $\mathcal{FV}$ for *faulting the signer* or *faulting the verifier*, respectively. Both scenarios, described in detail in Section 3.4, share that the injected faults target the checksum mechanism of the WOTS scheme to render it ineffective. We refer to the phase in which this fault is injected as *fault injection phase*, it is described in detail in Section 3.2. Also, in both scenarios, the adversary must perform a brute-force search to generate a signature for its malicious message. We refer to this phase as *brute-force phase* and it is described in Section 3.1.

3.1 Brute-Force Forgery of WOTS

In the following we assume that the checksum mechanism is not part of the WOTS scheme, i.e. is ineffective due to the injected fault. Without the checksum, a WOTS of the message digest m created by an entity A can be used to sign other message digests, e.g. to forge a signature for a malicious payload. This is possible as the hash chains can be advanced by repeatedly hashing the signature chunks. To be able to exploit this, the adversary needs to be in possession of a message digest r, s.t. $r_i \geq m_i \; \forall i$ ($r_i = \mathcal{N}(r, i)$ and $m_i = \mathcal{N}(m, i)$, see Section 2.1). The forged signature behaves as if A had signed r using its secret key. Finding a message which maps to such a message digest r is only possible through brute-force search, due to the preimage resistance of the underlying hash function. The number of trials that is necessary for an adversary to succeed in such a search is analyzed in Section 4.1. Section 6.1 reviews the means by which an adversary can efficiently perform the brute-force search.

3.2 Fault Attack on WOTS Checksum Chains

If the adversary is in possession of a malicious message digest r, s.t. $r_i \geq m_i \; \forall i$, the checksum of r will always be lower than that of m. The checksums cannot be equal as this would imply that all chunks of m and r are equal. We disregard the case where the adversary selects its malicious message to be equal to the original message, as this would be of no benefit. And, further, if the digests r and m are equal but not the messages, this would resemble an infeasible second preimage attack.

However, for some checksum chunks $r_j = \mathcal{N}(\mathcal{C}(r), j)$ and $m_j = \mathcal{N}(\mathcal{C}(m), j)$, $r_j \geq m_j$ may still hold. For these, the adversary can simply reuse or advance chains of the signature of m for her forgery. But, if $r_j < m_j$, the adversary must know prior nodes of the OTS checksum hash chain. Recovering prior nodes by inverting F is impossible as it is based on a cryptographic hash function. To overcome this issue, we instead propose a fault attack: The injected fault shall force a node to a lower level on the chain than required by the respective checksum chunk. Consider a value $v \in [0, w-1]$ for checksum chunk $m_j = v$. Then, either the corresponding secret key node sk_j (during signing) or the signature node sig_j (during verification) is advanced v or $w - v - 1$ times, i.e. $\mathcal{F}^v(sk_j)$ or $\mathcal{F}^{w-v-1}(sig_j)$. Our fault attack forces the implementation to use values smaller than the actual v, or $w - v - 1$, respectively. If the signing entity is attacked, prior nodes than the actual signature nodes are revealed. If the verifying entity is attacked, a correct public WOTS key is derived from nodes too far progressed. With the fault, the adversary is able to forge a valid WOTS for r.

We describe both attack variants in Section 3.4. For our theoretic analysis in Section 4.1 we assume that the adversary is able to completely skip the checksum calculations. In this case we do not need to care about individual checksum chunks and whether we can forge them or not. We refine this by limiting our attacker capabilities to skip or tamper single or multiple calculations in the theoretic analysis in Section 4.2. We show the practicability of our fault attack in Section 6.

3.3 Faulting WOTS to break LMS, XMSS, and SPHINCS$^+$

So far we have established how an adversary can forge a WOTS signature with fault injection. This section establishes that faulting WOTS is sufficient to break any of the HBS algorithms introduced in Section 2.2 and describes the attacks an adversary can mount on the respective schemes.

For the single tree variants of LMS and XMSS, the adversary is limited to attacking the only WOTS instance within these schemes, the one signing the actual message digest. A successfully forged WOTS signature is also valid for LMS and XMSS, as the Merkle tree in those schemes only authenticates the WOTS public key.

For the many-time signatures HSS and XMSS-MT, the adversary has more possibilities to mount an attack. If she chooses to attack the lowest WOTS, which signs the actual message, the attack is equivalent to the attack on a single tree scheme described above. However, choosing one of the intermediate WOTSs, which authenticates the root of the respective lower Merkle tree, allows an adversary to sign arbitrary malicious messages. This is because, if the attack on an intermediate WOTS succeeds, the adversary gains the capability to forge a signature for a root node of a lower Merkle tree. Once such a signature is forged, the adversary can arbitrarily construct an entire tree and is therefore in possession of a secret key, which can be used to sign (a limited amount of) arbitrary messages. For the brute-force phase we propose to use the topmost authentication node of the targeted intermediate tree as a counter to efficiently search for suitable root node candidates.

This also applies to the stateless signature scheme SPHINCS$^+$. The only difference between attacks on SPHINCS$^+$ and attacks on HSS and XMSS-MT is, that SPHINCS$^+$ uses FORS instead of WOTS to sign the actual message. However, this structural difference does not impact the adversary's capabilities to forge an entire Merkle tree.

3.4 Attack Variants

The two scenarios to which our attack applies, faulting the signer or verifier ($\mathcal{FS}$ or $\mathcal{FV}$), differ in the order in which the fault injection and brute-force phase take place.

Faulting the signer ($\mathcal{FS}$). In case of the $\mathcal{FS}$ scenario, the message and therefore the digest m is only known to the adversary after the signature was generated. Nevertheless, the adversary manipulates the WOTS checksum mechanism during signing. The general goal is to force the signer to not advance any checksum hash chain up to the needed signature node, i.e. manipulating $\mathcal{F}^{v}(sk_j)$ to $\mathcal{F}^{v'}(sk_j)$, where $v' < v$. This reveals nodes which need to be kept secret. Depending on the adversary capabilities described in Section 3.5, we show in Section 4.2 that there are different strategies to achieve this outcome.

The fault was successful, if the result is a tampered signature revealing e-nough prior nodes in the checksum hash chains. To forge a valid signature for the malicious payload, the tampered signature is used as an input for the brute-force phase. Here, the selected fault strategy also has an impact on the probabilities for finding a message digest which is suitable to forge a signature.

The malicious payload is forwarded with the forged signature to the victim for verification, e.g. during a secure boot or firmware update. Since the adversary crafted a dedicated payload for the tampered signature, the victim's verification of the message with the public key stored on the device yields a valid signature.

Faulting the verifier ($\mathcal{FV}$). In the $\mathcal{FV}$ scenario, the adversary is able to collect a set of signatures. These signatures are used as an input for the brute-force phase. Depending on the faulting capabilities of the adversary, the success probability of the brute-force phase, and therefore also the computational cost, vary.

During the fault injection, the adversary tries to force the verifier to not advance a checksum hash chain as determined by the respective checksum chunk, i.e. manipulate $\mathcal{F}^{w-v-1}(sig_j)$ to $\mathcal{F}^{o}(sig_j)$, where $o < w-v-1$. A straightforward approach for the adversary is to manipulate the victim, s.t. $o = 0$. In this case, the chain calculation of a checksum chunk is skipped entirely and the sig_j node of the forged signature is forwarded directly to the computation of the WOTS public key candidate. To achieve verification, the adversary sets sig_j to the top value of the respective chain, s.t. the correct public key is computed. In Section 4.2, we evaluate both relaxed assumptions on the adversary, where setting $o = 0, \forall j$ is possible and more constrained assumptions, where only individual checksum hash chains are (partly) skipped. As described above, these scenarios imply different degrees of freedom for the brute-force phase.

The malicious payload and the forged signature are forwarded to the target device for verification. To trick the verifier into accepting the invalid signature containing invalid OTSs for the checksum, an adversary applies the fault attack as described above. The fault injection was not successful, if the verifier advances this hash chain too far and calculates an invalid compressed OTS public key, which fails verification. If the fault injection was successful, the verifier derives the correct WOTS public key, the signature is verified as valid, and the malicious payload is accepted by the target device.

3.5 Adversarial Model

In this section, we introduce the faulting and brute-force capabilities of the adversary.

Faulting capabilities. A fault attack has the purpose of manipulating the control or data flow of an application to achieve an outcome that is desired by the adversary. Typical fault attacks we deem applicable to this work are clock and voltage glitching, electromagnetic fault injection (EMFI), laser fault injection (LFI) or software-based hardware attacks like Rowhammer. To simplify analysis, we condense all these attacks into two basic fault models. Please note, that this is not sufficient to fully analyze an implementation. To do so, fault models specific to the underlying hardware need to be derived and used for analysis of the exact data and control flow.

The first fault model we deem reasonable allows an adversary to *skip a single instruction.* This fault model is frequently reported in literature and has been demonstrated on various embedded devices [OSS17, GTSC, O'F19].

The second fault model allows the adversary to tamper data. More precisely, we assume that the adversary is able to *inject single or multiple bit-flips* into registers or memory [SZK$^+$18, FKK$^+$22]. By applying both fault models and showing vulnerable spots within the HBS implementations (Section 6), we want to highlight the general applicability of our attack to several devices in different environments and scenarios.

Brute-forcing capabilities. As this attacks bears some computational complexity, we need to evaluate its feasibility depending on the adversary's capabilities. To do so, we base our categorisation on [Aum19], which classifies security strengths below 100 bits as *weakened*, and below 80 bits as *broken.* This is commonly used in similar scenarios like side-channel analysis [VCGS13, HMU$^+$20]. In Section 6.1, we evaluate different hardware platforms (CPUs, GPUs, or ASICs) to give an estimate for the economic costs connected to this attack.

4 Probabilistic Analysis

In the previous section, we established that the complexity of the fault attack, and the complexity of the brute-force search for a suitable message digest to forge a signature are connected. In the following section, we first analyze the computational complexity for the brute-force phase when assuming that the checksum is rendered completely ineffective by the fault attack (Section 4.1). We refine these probabilities and the cost of the attack wrt. the faulting capabilities of the attacker in Section 4.2.

4.1 Probabilities

For the attack, the adversary needs to find a digest whose signature is forgeable by using a set M of signed random message digests. We assume the adversary has intercepted the set M of signed digests $m \in M$. As the attacker does not have an influence on the digests contained in M, its capabilities are those of a random message attack (RMA). She now performs the calculation of digests r of messages that are usable for the attack. The set of these trials is R. Even if the adversary has a specific target message, e.g. in the form of a binary, an infinite number of potential forgery targets can be generated by appending a counter to the payload. Among others, this principle was also used in [BHRVV] to efficiently generate a vast amount of different message digests. If it is not possible to append a counter to the selected message, an attacker can exploit the fact that for LMS and XMSS the message is digested using a method called randomised hashing. The hashing instance is initialised with a seed chosen by the signer. An attacker can therefore choose arbitrary values. For SPHINCS$^+$ a different approach is used, with similar capabilities, which is described in more detail at the end of this section. We thus assume that, if needed, the adversary can generate any amount of candidate digests, only limited by its computational resources. In the following, we describe the attack scenario for the adversary goals of universal forgery (UF), selective forgery (SF), and existential forgery (EF). These goals were also used in [GBH18] to evaluate their attack.

To model the probability, we need to know the distribution of values the signed message can take. More precisely, we require the distribution of the message digest, as the message is always hashed prior to signing. We are not interested in weaknesses of the underlying hash functions, therefore we assume that $\mathcal{F}$ behaves like an oracle with uniformly randomly distributed output.

Universal forgery (UF). An UF is a the strongest forgery attack as it enables an adversary to sign any given digest r. When applied to WOTS, it is necessary for the attacker to possess a valid signature of a message digest m that consists of all zeros – which is rarely the case. However, if the adversary has obtained such a signature, she has obtained all OTS secret keys. Hence, each hash chain can be advanced to an arbitrary node, signing any message. The probability for a single hash chain of length w to be equal to zero is $\frac{1}{w}$. The probability that this is the case for all hash chains is

$$p_{UF} = \left(\frac{1}{w}\right)^{l_1} = 2^{-8n}.$$

Given a set of M validly signed message hashes, the probability that one of them is a zero-hash can be modeled by the CDF of a geometric distribution with parameter p_{UF} as following

$$Pr_{break}[M, R] = 1 - (1 - p_{UF})^{|M|}.$$

The adversary has no possibility to increase the overall probability as its brute-force set R has no influence. As $n \in [16, 24, 32]$ – for the NIST security levels of one, three, and five, respectively – the success probability to achieve UF-RMA when our attack is applied only to WOTS is infeasible. By extending our attack to HBS schemes with multiple trees, we show how an attacker can still achieve UF-RMA within certain constraints. This extension is described in Section 3.3.

Selective forgery (SF). In case of a SF, an adversary chooses a fixed digest r before gaining knowledge of the set M. Based on [GBH18], we model this scenario as follows: To maximize the likeliness that the chosen r is a forgery candidate for the unknown set M, the adversary chooses a threshold $b \in [0, w-1]$. Now, an r where for each chunk r_i it holds that $r_i \geq b$ is pre-calculated. The attack succeeds with such a chosen r, if M contains a message hash m where for each chunk m_i holds $m_i \leq b$. In this case, the adversary has knowledge of an r for which holds that $r_i \geq m_i, \forall i$. Thus, m can be misused to forge a signature for

r. Due to the equally distributed output of the hash function, the probabilities that $r_i \geq b$ and $m_i \leq b\ \forall i$ are given as

$$p_{SF_{\geq b}} = \left(\frac{w-b}{w}\right)^{l_1},\ p_{SF_{\leq b}} = \left(\frac{b}{w}\right)^{l_1}.$$

Each of the two cases applied to the whole set, i.e. $\exists m \in M \,|\, m_i \leq b$ and $\exists r \in R \,|\, r_i \geq b\ \forall i$, can again be modeled as the CDF of a geometric distribution dependent on the size of the set. Thus, the joint probability of both occurring is

$$\begin{aligned} Pr_{break}[M,R] = &\left(1-(1-p_{SF_{\leq b}})^{|M|}\right) \\ &\cdot\left(1-(1-p_{SF_{\geq b}})^{|R|}\right). \end{aligned}$$

The selection of the threshold b constitutes a trade-off, as a higher b leads to a higher $p_{SF_{\leq b}}$ allowing for a smaller set M, but at the same time raises the necessary pre-computation for the set R as $p_{SF_{\geq b}}$ drops. In the case that $\exists m \in M \,|\, m_i \leq b\ \forall i$ a signature m can still be leveraged for the forgery, if $\exists m \in M$ and $\exists r \in R \,|\, m_i \leq r_i\ \forall i$. Thus the actual probability is *at least* the above.

The SF scenario corresponds to cases where the computation of the forgery candidate needs to occur before the attacker gets access to m. We deem this as less relevant and therefore do not further investigate this scenario within this work.

Existential forgery (EF). In case of EF, an adversary succeeds in signing one arbitrary digest. To achieve this when given set of signed digests M, the adversary performs a calculation of forgery candidates r for each $m \in M$, i.e. the size of both sets are equal: $|M| = |R|$. The probability that one chunk r_i of the candidate is greater or equal to the corresponding chunk m_i of the message m can be described using the law of total probability

$$\begin{aligned} &Pr\left[r_i \geq m_i\right] = \\ &= \sum_{x=0}^{w-1} Pr\left[r_i \geq m_i | m_i = x\right] \cdot Pr\left[m_i = x\right] \\ &= \sum_{x=0}^{w-1} \left(\frac{w-x}{w}\right)\frac{1}{w} = \frac{w+1}{2w}. \end{aligned}$$

This leads to the overall probability for l_1 chunks of r being larger than m with $|R|$ number of trials:

$$Pr_{break}[M,R] = 1 - \left(1 - \left(\frac{w+1}{2w}\right)^{l_1}\right)^{|R|}. \tag{3}$$

The results for different parameters are plotted in Figure 3. As we assume for each forgery trial to draw an unseen m, we model the probability for each trial as independent event. While this case allows an exact calculation of the corresponding probabilities, it can only be used to roughly estimate the order of complexity for cases where $|M| = I$ and $|R| \gg I$. Hence, it remains unclear how many trials are required in the adversaries scenario and a more exact representation of the probability is needed.

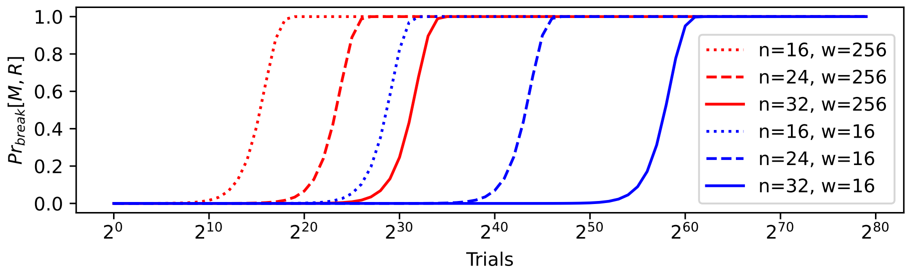


Fig. 3: The success probability of an EF-RMA with $|M| = |R|$.

A different approach is to calculate the probability of a forgery for each m individually. For a certain m we can calculate the probability that all l_1 chunks r_i of the candidate are greater or equal to the corresponding chunks m_i:

$$Pr\left[r_i \geq m_i, \forall i\right] = \prod_{i}^{l_1} \frac{w - m_i}{w}.$$

In contrast to the previous scenarios, this probability is dependent on m. It is thus not possible to derive a general p_{EU}. The probability of breaking a non-fixed message m using a set R is the sum of probabilities for all possible m, each of which occurs with the same probability:

$$\begin{aligned} Pr_{break}[m, R] &= \sum_{m \in M} (1 - (1 - Pr[r_i \geq m_i, \forall i])^{|R|}) \cdot Pr[m] \\ &= \sum_{m \in M} (1 - (1 - Pr[r_i \geq m_i, \forall i])^{|R|}) \cdot \frac{1}{w^{l_1}}. \end{aligned}$$

For a set M, the overall probability thus becomes

$$Pr_{break}[M, R] = 1 - \left(1 - Pr_{break}\left[m, R\right]\right)^{|M|}. \tag{4}$$

To calculate the probability $Pr_{break}[m, R]$, all valid digests $m \in M$ need to be evaluated. Due to the amount of possible outputs of a cryptographic hash function, this is infeasible to compute. Therefore, we propose to approximate the expected probabilities with the help of simulations. If the values from Figure 3 are taken as a reference point to estimate the complexity for simulating the attack, it becomes obvious that a high resource usage is required. For example, the experiment with $n = 32, w = 16$ might require up to 2^{60} trials. Further, we would have to run the experiment a significant number of times to draw conclusions from it and the simulation only allows to draw conclusions for the number of trials performed.

To circumvent this issue, we instead simulate $Pr_{break}[M, R]$ for $|M| > 1$. This reduces the computational effort and, thus, allows to use general-purpose computing equipment. With these results we can approximate $Pr_{break}[m, R]$, i.e. $|M| = 1$ as

$$Pr_{break}\left[m, \frac{R}{|M|}\right] = 1 - \sqrt[|M|]{1 - Pr_{break}[M, R]}. \tag{5}$$

Figure 4 shows our results. Please note that the analysis of the $w = 16$, $n = 32$ parameter set exceeded our available computing resources and is therefore omitted. For the parameter set of $w = 16, n = 24$, we have run our simulations with $|M| = 131072$. Due to the high count of $|M|$ and limited computing resources, we selected 4096 messages from M, for which the brute-force search had the highest success probability based on Equation (3) – effectively reducing the input to the brute-force search by 32. Hence, Pr_{break} for $w = 16, n = 24$ is *at least* as high as given by the results of our simulations. To reflect this in the plots, we marked the count of the message set with $|M^*|$.

Extending existential forgery to universal forgery. The results shown so far demonstrate that, if only a single WOTS signing a message is targeted, an adversary can only achieve EF with reasonable high probabilities. This changes if the attack is applied to any of the multi-tree algorithms, i.e. the HSS variant of LMS, XMSS-MT, and SPHINCS$^+$. By exploiting the dependency between trees, the adversary is able to extend its forging capabilities such that UF can be achieved. Section 3.3 established that an adversary may target WOTS instances that authenticate sub-trees. This affects the brute-force complexities slightly. The input to the signing or verifying operation is no longer a message, which can be chosen freely, but the root node of a sub-tree. During the brute-force phase, the adversary must generate a new sub-tree with a suitable root node. For this, the secret key (a seed which is used to generate all the leaf nodes) is chosen freely. Then, the adversary constructs the tree from the secret key and divides it into signature and authentication nodes (see Section 2.2). The top-most node of the authentication path can be replaced with a counter which is iterated until a suitable root node is found. The difference to attacking a WOTS instance signing a message is that the adversary is now capable of authenticating a key pair and is in possession of the secret key. Therefore, the adversary gains the possibility to sign messages without any additional effort. If a message-signing WOTS instance is attacked, every new message requires a new brute-force phase. Hence, the attack is extend from EF to UF.

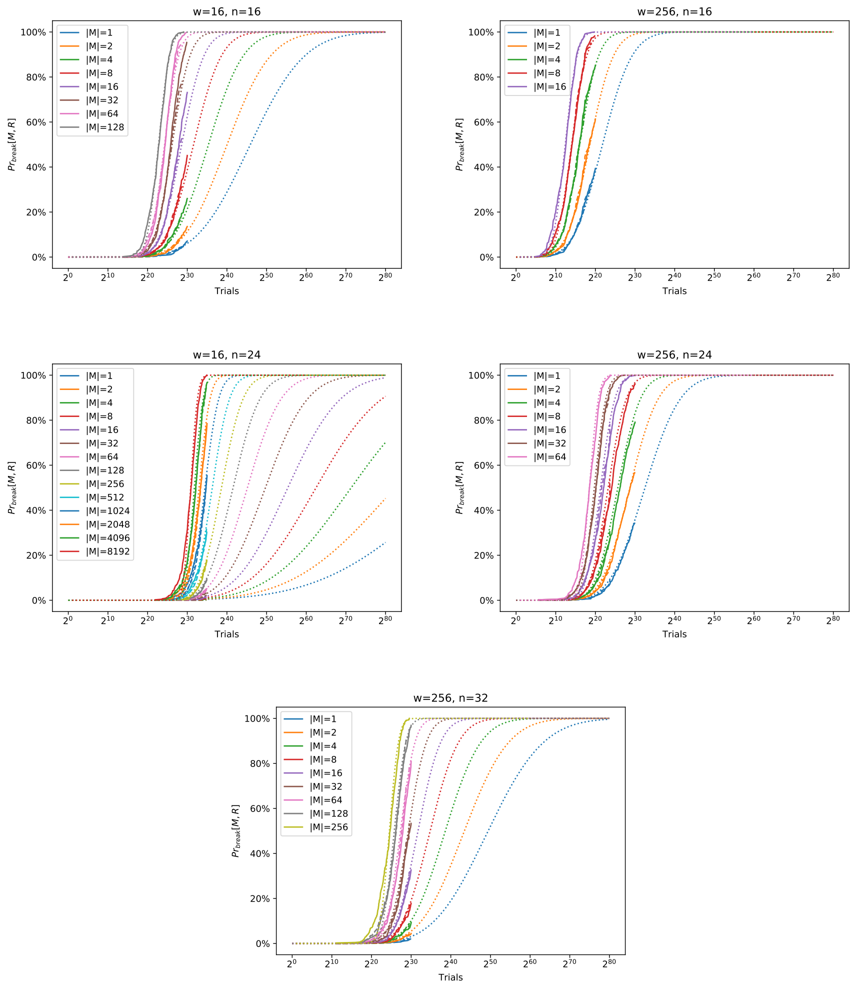


Fig. 4: Simulated (solid line), and approximated (dotted line) probability of finding suitable hash for different Winternitz parameters w, hash output lengths n, and signature set sizes $|M|$, with respect to the number of trials $|R|$. All results for a certain set size $|M|$ are plotted using the same color. The dashed lines represent the approximations obtained from the respective maximum $|M|$. The solid lines show actual simulation results to verify the estimated data.

4.2 Probabilities wrt. adversary capabilities

In this section, we empirically derive the theoretic probability that an adversary can produce a valid forgery with a digest r derived from the brute-force phase by means of fault injection. The general probability to find such a digest r can be obtained from Figure 4. Recall, that in Section 4.1 and Figure 4, the assumption was that the complete checksum computation can be skipped. Now the probabilities that r is suitable for a forgery in more constrained fault models are analyzed. Recall that for a forged signature to be accepted, each chunk of the faulted checksum $\mathcal{C}(r)$ needs to be larger or equal to the corresponding chunk of $\mathcal{C}(m)$. Therefore, suitable in this context means that faults corresponding to certain models can be used to manipulate the checksum in this manner. We define this probability as Pr_{valid}. Obviously, this probability varies depending on the adversary's fault injection (FI) capabilities.

Two properties factor into Pr_{valid}: the type of fault attack and the number of independent faults injected during one operation, either during signing or verifying. We introduced the general FI capabilities of the attacker in Section 3.5. We simplified our evaluation by assuming two fault models, control flow and data corruption. For control flow corruption we only consider single instruction skips, for data corruption we analyze single and double bit-flips. We assume bit-flips for arbitrary positions with no positional constraints. Further, in the case of double bit-flips, both are independent from each other and behave as two single bit-flip faults. From these fault models, we derive three concrete fault scenarios, which are listed in the following.

In Section 3.4, two variants of our attack were introduced, $\mathcal{FS}$ for applying the fault attack to the signer, and $\mathcal{FV}$ for applying the fault attack to the verifier. For the $\mathcal{FS}$ scenario, the adversary needs a strategy to fault the signing operation without knowledge of the message and its signature, s.t. the probability to find a forged signature in the brute-force phase is as high as possible. For all listed scenarios we list the highest achievable probability with the respective strategy in Table 2. In the $\mathcal{FV}$ scenario, the attacker is already in possession of a forged signature obtained in the brute-force phase and therefore knows how the checksum computation needs to be manipulated. Therefore, for the listed probabilities in Table 1 it is not necessary to differentiate between different strategies.

Single hash chain skip The attacker skips the calculation of one OTS checksum hash chain by means of a single instruction skip. In our practical analysis in Section 6, we show that this is a reasonable assumption. We assume that the adversary is able to skip one chosen checksum hash chain precisely. The exact point in time of individual checksum hash chains depends on m and therefore is not constant in time. However, in the $\mathcal{FV}$ scenario, the adversary knows the order of operations as r is known. In case of the $\mathcal{FS}$ scenario, the adversary could obtain such informations via a side-channel inspection. If the hash chain calculation is skipped, the chain will not be advanced by the signer or verifier at all, but execution stalls with the input node. Therefore, the input node is used as-is for the further execution of the algorithm.

Single bit-flip The attacker corrupts the value of one of the checksum chunks with a single induced bit-flip. Ideally, this manipulates the checksum operation, s.t. instead of advancing by v steps, i.e. $\mathcal{F}^v$, the chain is only advanced by $v' < v$. In contrast to hash chain skips, timing might be less of an issue as the data can be targeted while stored in memory.

Double bit-flip The attacker corrupts the value of at least one of the checksum chunks with two induced bit-flips. We do not differentiate between cases where both bit-flips target the same chunk and cases where the bit-flips apply to different chunks. In contrast to the single bit-flip scenario, a double bit-flip has more possible outcomes in terms of how the checksum chunks are manipulated. Therefore, $v' < v$ can hold for one or two checksum hash chains.

Table 1: The average probability Pr_{valid} in case of $\mathcal{FV}$ for a single suitable hash r depending on the attacker capabilities.

Fault type	Scenario	w=16		w=256		
		n=16	n=24	n=16	n=24	n=32
Control flow corruption	Single hash chain skip	54.1 %	44.1 %	46.8 %	39.6 %	51.0 %
Data corruption	Single bit-flip	54.1 %	34.8 %	46.8 %	39.6 %	51.0 %
	Double bit-flip	89.1 %	65.8 %	81.2 %	72.0 %	83.8 %

Table 2: The average probability Pr_{valid} for the most suitable fault locations in case of $\mathcal{FS}$ for a single suitable hash r depending on the attacker capabilities.

Fault type	Scenario	w=16		w=256		
		n=16	n=24	n=16	n=24	n=32
Control flow corruption	Single hash chain skip	54.1 % $r_j \mid j = 0$	34.8 % $r_j \mid j = 0$	46.8 % $r_j \mid j = 0$	39.6 % $r_j \mid j = 0$	51.0 % $r_j \mid j = 0$
Data corruption	Single bit-flip	54.1 % bit 8	28.0 % bit 8	46.8 % bit 11	17.3 % bit 11	51.0 % bit 12
	Double bit-flip	56.6 % bit 8, 7	29.3 % bit 8, 7	48.0 % bit 11, 5	22.5% bit 11, 10	51.4% bit 12, 4

Table 1 shows the average Pr_{valid} for the scenarios of $\mathcal{FV}$. A value of $Pr_{valid} = 54.1\%$ for $\mathcal{FV}$, $w = 16$, $n = 16$, and *single hash chain skip*, reads as "slightly more than half of all digests found in the brute-force phase described in Section 4.1 can be realized in scenarios, where only one fault can be applied, s.t. a single hash chain calculation is skipped". Obviously, an adversary can determine directly from r, whether an attack will succeed within a certain fault scenario. Therefore, the fault injection phase is only performed for digests r for which $Pr_{valid}(r) = 1$. In the $\mathcal{FS}$ scenario, Table 2 additionally shows which strategy an adversary needs pursue to achieve the highest probability Pr_{valid}. A value of $Pr_{valid} = 54.1\%$ with $r_j|j = 0$ for $\mathcal{FS}$, $w = 16$, $n = 16$, and *single hash chain skip*, reads as "if during signing, a checksum chain with index $r_j|j = 0$ is skipped, slightly more than half of all digests found in the brute-force phase described in Section 4.1 can be used to forge a signature". For the fault type of data corruption we list Pr_{valid} along the bits that needs to be targeted during signing to achieve this probability, e.g. for the parameter $w = 16$ and $n = 16$ the adversary has the highest probability of 54.1 % or 56.6 % if fault attacks with single-bit flips target bit 8 or with double bit-flips target bit 8 and 7, respectively. This is due to the fact that the brute-force phase described in Section 4.1 does not have any constraints on the checksum and its chunks, but only on the message chunks ($r_i \geq m_i \; \forall i$). These constraints are introduced by the fact that candidates found in the brute-force phase can only used for forgery with a probability of Pr_{valid}.

The analysis of the probabilities in general and the strategies for $\mathcal{FS}$ is based on the simulation results displayed in Figure 4. For each pair of (r, m) we applied any possible fault for the three listed fault scenarios. We applied the fault to the checksum of m or r for $\mathcal{FS}$ or $\mathcal{FV}$, respectively. We calculated Pr_{valid} by dividing the number of suitable sets (r, m) for forgery by the total number of generated candidates r which fulfill the constraints of $r_i \geq m_i \; \forall i$.

5 Countermeasures

To protect HBS implementations against the presented attack, different measures are applicable depending on the two attack variants, $\mathcal{FS}$ and $\mathcal{FV}$. In the case of $\mathcal{FS}$, hardening the implementation is straightforward. As a consequence of the tampered signature generation, the signer generates an invalid signature. This weakness of the attack can be used to design a countermeasure (CM). If the signer verifies the signature after generation, the attack will be detected. Since the cost of verification is minimal compared to signature generation, this step can easily be added by the signing entity. In the case of $\mathcal{FV}$, the countermeasures are more diverse and costly. We describe their design in the following sections. However, due to the more complex approach, we evaluate their efficiency and effectiveness in detail in Section 6.

Hash chain length calculations. Any error introduced into the calculation of the hash chain length can lead to a wrongly calculated, but potentially exploitable, hash chain length value. Repetition and comparison of the calculated hash lengths allows any tampering to be detected. The cost of this countermeasure is small, as this part of the algorithm is negligible in terms of overall performance.

Hash chain calculations. The countermeasures for the hash chain calculations can be divided into two independent levels: skipping partial and full hash chain calculations.

The attack vector of a *partial skip* of a hash calculation can be avoided by using a memory comparison of the input and output buffers. This will detect if the hash operation was skipped and thus render the attack vector ineffective. This countermeasure only needs to be performed during the first iteration, as an adversary must skip from the first iteration onwards. This is due to the fact that the iteration index is an input to each hash step calculation. To ensure successful verification, the verifier must be tricked to combine the malicious checksum node with the correct iteration index. Therefore, it is not possible to swap hash steps and the skip must be introduced from the first iteration onwards.

The *complete skip* of the hash chain calculation can be countered by assuring that at least one hash chain calculation is performed. Combined with the countermeasure to disable a partial skip of the hash calculation, this makes this attack vector impossible to execute. In practice, an implementation can simply return the iteration counter. The calling function compares the returned value with the maximum value for the iteration. If it does not match, a fault has been introduced and execution is aborted.

WOTS+C. WOTS+C is designed to compress WOTSs [KHRY22] as introduced in Section 2.1. The key idea behind this study is to skip the checksum chains in favor of a checksum with a fixed value. This makes control flow attacks to skip a checksum chain no longer feasible. The applicability of data errors needs to be investigated, as well as any impact on the brute-force phase. Operations such as checksum comparison may be suitable targets for FI and must therefore be hardened.

6 Attack in Practice

In this section we describe a real-world scenario to demonstrate the practicability and severity of our attack. The target is an embedded device based on the commonly used ARM Cortex-M4 processor. The attacker's goal is to run malicious firmware on the embedded device. To protect the firmware execution on the device against tampering from physical adversaries, secure boot is used directly after power-up to verify any firmware after loading it into the internal memory and before execution. For verification one of the hash-based signature schemes is used. The secure boot implementation is based on MCUboot [mcu]. We assume that the attacker has physical access and therefore is able to modify or exchange the off-chip stored firmware before it is loaded into the internal memory and to execute a fault attack.

Fault Model and Hardening. If MCUboot is used, the secure boot implementation is partially hardened against FI attacks [mcu]. The scope of the hardenings is to, for example, protect against an instruction skip. Therefore, instruction skip fault attacks that target the generic secure boot flow, e.g. ensuring that only valid images are booted, will not be successful. The cryptographic implementations are only partially hardened with similar countermeasures [Ban] and may therefore still be vulnerable to FI attacks.

PQ Secure Boot. To fulfill the requirements of a post-quantum secure boot, the targeted embedded device verifies each stage using a HBS scheme instead of classical asymmetric cryptography. Please note that MCUboot does not yet support PQC schemes, but plans to do so [Bro]. We select the algorithms and the respective parameters based on the results in Figure 4, related research works [KPC$^+$] and public available information on embedded devices, which employ HBSs for secure boot or firmware updates or plan to do so [Phi22]. The probabilistic analysis in Section 4.1 has shown that – for the analyzed parameters – the parameter set of $w = 16$ and $n = 24$ has the highest brute-force complexity. Therefore, we deem it relevant to be investigated within this scenario. As we select these parameters, we assume either LMS or XMSS with a single tree, reflecting the worst case for the attacker based on our results. In [KPC$^+$], HSS and SPHINCS$^+$ with 192-bit key length, $w = 256$, and three or five Merkle trees, respectively, is considered as relevant for UEFI secure boot. In [Phi22], SPHINCS$^+$ with 128-bit key length is reported as suitable for the secure boot of an embedded device. As the tree structure is not specified, but a reduced maximum signature count is requested, we assume a similar tree structure as in [KPC$^+$], and set the Winternitz parameter as $w = 16$ based on the statement in [Phi22] that performance is a constraint.

In summary, this results in these distinct algorithms and parameter sets for our practical analysis: LMS or XMSS with $w = 16$ and $n = 24$; HSS with three Merkle trees, $w = 256$ and $n = 24$ [KPC$^+$]; SPHINCS$^+$ with five Merkle trees, $w = 16$ and $n = 16$, and $w = 256$ and $n = 24$ [KPC$^+$, Phi22].

6.1 Brute-force Forgery of WOTS

As described in Section 3.1, the brute-force phase requires at least one WOTS message-signature pair as input, in this scenario we use the more suitable term of firmware-signature pair. Within this attack, the adversary has access to the external memory, which contains firmware and the corresponding signature. Because the device receives updates, the adversary is even capable to collect several pairs for the brute-force. To reflect different update intervals we analyze the scenario with the assumption that the adversary can collect firmware-signature pairs with a count out of $[1, 10, 100, 1000]$.

By applying our fault injection attack, a verifier is tricked to assume a malicious signature as valid. Before the actual fault attack, the attacker must forge this signature. This brute-force process can happen "offline" and "off-site", i.e. there are no strict timing requirements. Depending on the HBS scheme the adversary achieves existential or universal forgery as described in Section 3.3 and Section 4.1. In practice, the efficiency of signature forging boils down to the number of hash calculations per timespan, i.e. the hash rate, the attacker

Table 3: Hash rates of SHA-256 for different platforms.

Hardware	Type	Hash rate
Intel i7-9700K [Son19]	CPU	299 MH/s
Nvidia RTX 3090 [Cro20]	GPU	9.71 GH/s
Nvidia RTX 4090 [Cro22]	GPU	22.0 GH/s
Antminer S19 XP [Bit22]	ASIC	140 TH/s

can achieve. Table 3 shows different platforms with their respective hash rates. For central processing units (CPUs) and graphics processing units (GPUs), the benchmarks were performed with hashcat.

In comparison to CPUs and GPUs, application-specific integrated circuits (ASICs) achieve the best performance. However, we deem ASICs less relevant s-ince their dedicated design would make them very expensive. GPUs are, however, attractive due their combination of high performance and flexibility. The attack-er can easily gain access to many devices, e.g. from cloud computing providers. We selected a single Nvidia RTX 4090 GPU to estimate the time required for a brute-force search for the selected parameters. The results in Figure 5 show that, for all three parameter sets based on a multi-tree structure, a forgery suc-ceeds ($Pr_{break} \geq 90\%$) in less than an hour. We achieve these results even if an adversary has only access to a single firmware-signature pair. While the effort is significantly larger for single tree structures ($d = 1$), it is still feasible, e.g. if multiple GPUs are available.

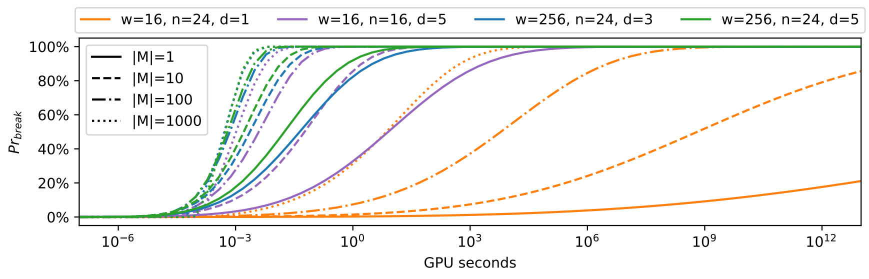


Fig. 5: Cost estimation for brute-force forgery of the selected parameter and structure with a single GPU. Estimations are displayed for the number of firmware-signature pairs for the values of 1, 10, 100, and 1000.

6.2 Fault Attack on Hash-Based Signatures

Having the possibility to forge a signature for a malicious firmware image brings the attacker one step closer to the goal of executing malicious code. The miss-

Table 4: Number of instructions that will lead to a successful verification of an invalid signature if only one is skipped.

	Optimization	CMs	Generic FI			WOTS FI		
			(1)	(2)	(3)	(1)	(2)	(3)
LMS	-O2	No	20	-	15	32	-	27
		Yes	1	-	0	1	-	0
	-Os	No	18	-	18	33	-	33
		Yes	2	-	0	2	-	0
XMSS	-O2	No	6	2	1	17	13	12
		Yes	5	1	0	5	1	0
	-Os	No	13	10	0	21	18	8
		Yes	13	10	0	13	10	0
SPHINCS$^+$	-O2	No	2	1	1	7	6	6
		Yes	1	0	0	1	0	0
	-Os	No	5	4	0	10	5	5
		Yes	5	4	0	5	4	0

ing piece is to inject a fault to circumvent the signature verification. To assess the possibility of a successful fault attack we search for weak spots within the reference implementations of LMS, XMSS, and SPHINCS$^+$. We base our analysis on an extensive emulation of all possible faults based on the instruction skip fault model using ARCHIE [HGA$^+$21]. The emulation is performed for the ARM Cortex-M4 processor. We analyse two different scenarios reflecting the two different approaches to optimising for performance or size and their impact on fault injection resilience. Therefore, the firmware is compiled for the two scenarios with either of the two optimisation levels: *-O2* and *-Os*.

We include both generic and WOTS-specific fault attacks in the analysis. Generic fault attacks are more straightforward for an attacker, as no brute-force phase is required. However, systems that require physical security can easily be protected against such attacks, without a detailed understanding of the underlying algorithms. In the following, we will demonstrate this by hardening a reference implementation with generic countermeasures. In contrast, we show that mitigating WOTS-specific attacks is not as straightforward. One needs to be aware of them and the vulnerable points they cause during execution. We demonstrate that generic countermeasures do not consider these points and that WOTS-specific attacks require a more thorough understanding in order to design integrated countermeasures. We apply the countermeasures proposed in Section 5 and evaluate their effectiveness and efficiency. The comparison of both types of attacks allows to understand their different leverage points. Furthermore, this analysis demonstrates the severity of an unprotected implementation and shows that countermeasures are feasible.

Table 5: Code size [B] of the signature verification routine for the different HBS schemes, optimisation levels, and CM tiers.

	Optimization	Code size	Code size increase by CMs (1)	(2)	(3)
LMS	-O2	2650	+206	-	+258
SHA2_10_256	-Os	2380	+164	-	+216
XMSS	-O2	4380	+126	+138	+162
SHA2_10_192	-Os	3870	+98	+114	+134
SPHINCS$^+$-s	-O2	10400	+106	+118	+134
sha2-128s	-Os	9550	+84	+100	+112

Table 6: Cycle count [cc] of the signature verification for the different HBS schemes, optimisation levels, and CM tiers.

	Optimization	Cycle count	Cycle count increase by CMs (1)	(2)	(3)
LMS	-O2	8.5M	+3.94k	-	+5.15k
SHA2_10_256	-Os	8.7M	+3.80k	-	+5.02k
XMSS	-O2	31.6M	+3.93k	+3.93k	+4.10k
SHA2_10_192	-Os	32.3M	+4.11k	+4.11k	+4.27k
SPHINCS$^+$-s	-O2	17.9M	+15.3k	+15.3k	+15.4k
sha2-128s	-Os	18.3M	+17.8k	+17.8k	+17.9k

The XMSS and SPHINCS$^+$ APIs provide a device with several ways to evaluate the result of the signature verification. As is common practice, the return value of the signature verification routine indicates whether an error has occurred or not. In addition, two other values, the message and its length, can be used to evaluate the result. In the event of an error, the message points to a array initialized with zeros and the corresponding length is set to zero. In the following, we will outline how these variants significantly increase the fault injection resilience. Hence, making fault attacks more difficult to execute. For this purpose, we group these variants into three enumerated categories with increased cost to the verifier: the verifier checks (1) the return code, (2) the return code and the message length, or (3) the return code, the message length, and the message. The original implementation of LMS does not support this functionality. To allow for a similar analysis, we add the ability to check the return code as well as the message. In the following this variant is labeled as (3).

Note that none of the reference implementations claim to be fault injection resilient. At the time of writing, there are no other HBS implementations available stating any countermeasures against fault attacks. Table 4 shows the overall fault injection resilience for the different scenarios. The count is the number of instructions where skipping one is sufficient to bypass the signature check. Therefore, a higher count corresponds to a lower resilience. Table 5 and Table 6 list the increased code size and execution time of the signature verification routine introduced by the variants for checking the returned values and the countermeasures.

Generic fault attack. Comparing the two stateful reference implementations, the XMSS implementation is more resilient to generic FI attacks. The main reason for this is the return of multiple values as described above and its internal structure. Most importantly, the comparison of the public key and the computed candidate directly triggers the writing of the return values. Furthermore, XMSS does not contain any bridging functions that may introduce exploitable weaknesses. However, despite the relatively high level of resilience, we still found vulnerable spots in XMSS that could be exploited with a generic FI attack. The existence of vulnerabilities depends on the level of compiler optimisation chosen. For the speed optimisation a countermeasure is needed to harden the comparison of the public key with its computed candidate. For all the experiments carried out, two measures were sufficient to protect this potentially fragile point. The first is to check that the length which is an input to the *memcmp* function is equal to zero. If this condition is met, an error is thrown and the execution is aborted. This is necessary because the *memcmp* function will always return zero for a length of zero despite the values contained in the two pointers. And the second part is to mark the returned value as volatile to allow for a repeated comparison of the returned value. The combination of these two measures effectively prevents tampering with this operation.

The LMS implementation differs fundamentally from the XMSS implementation. The major difference in terms of the fault injection resilience is that it only returns a single value to check the status of the signature verification. In general, the implementation contains more bridging functions, which has the effect that more measures to harden the implementation are required. As a result the required code size for the countermeasures listed in Table 5 is larger than for the other two implementations. The countermeasures required are similar to the hardening of the *memcmp* routine described above. As with the other two implementations, the comparison of the public key and the computed candidate must be hardened. In addition, for each returning bridging function, the check on the return value must be hardened by duplicated checks, and if it returns a Boolean value, it must be cast to an integer. The integer casting is required because we have experienced that for Boolean return values the compiler most often compares if the value is not equal to zero, resulting in false positives. Another necessary countermeasure is to initialise the error state with an initial error code. This way, any premature return will return the uncleared error statement and allow potential malicious tampering to be detected.

The SPHINCS$^+$ reference implementation is very similar to the XMSS implementation. The analysis showed a fairly resilient implementation. Nevertheless, SPHINCS$^+$ also requires a hardening of the public key comparison with its computed candidate. The hardening of this operation can be done with the same countermeasures as described above.

In conclusion, all three reference implementations of the HBS schemes can be hardened with simple measures so that there are no vulnerable instructions that could be skipped to lead to a successful verification of an invalid signature using a generic fault attack. This is reflected within the results in Table 4 showing that there are non weak spots left for a generic fault attack, if the proposed countermeasures are applied. Neither executing the generic attack nor designing countermeasures against it required any special knowledge of the running algorithms. The weak spots targeted by a generic fault attack are easy to spot for an attacker as well as the defender. Hence, an implementation is more likely to be resilient against a generic fault attack.

WOTS-specific fault attack. The results in Table 4 demonstrate that while an implementation can be hardened, there may still be potential vulnerabilities to a specific attack. In the worst case, including a WOTS-specific attack triples the number of vulnerable locations. Even worse, some implementations, such as the size-optimised XMSS with three return values, appear to be resistant to fault injection, but are not when a WOTS-specific attack is executed. We can therefore conclude that the generic countermeasures are effective, but do not protect against WOTS-specific attacks. However, this is different for the WOTS-specific countermeasures proposed in Section 5. When the countermeasures are used together with the three return value variant, all implementations for all scenarios are resilient against each of the two types of attack. In spite of the different code bases of the three reference implementations, all of them allow for similar countermeasure approaches without any loss of effectiveness. This is, of course, mainly due to the high degree of similarity between the three algorithms, which is also reflected in their implementations.

Effectiveness and efficiency of CMs. Due to the countermeasures applied, the implementations suffer in terms of performance and increased code size. The increase in execution time is listed in Table 6. Surprisingly, the performance is hardly changed by the countermeasures. Due to the adapted design of the countermeasures, a small impact was expected. But for all three implementations, the impact of the changed execution time is only about one permille or less. The impact on code size, listed in Table 5, is much more significant. The increase is in the range of one to ten percent. The additional size of the countermeasures is about 100 to 250 bytes, depending on the reference implementation, the level of optimisation and the number of return values. Due to the similarity of XMSS and SPHINCS$^+$, the implemented countermeasures are very similar and therefore the absolute impact on code size is comparable. The relative difference varies mostly due to a different implementation of the underlying hash function. In both relative and absolute terms, the LMS reference implementation has the largest code size increase due to the countermeasures. This is because this implementation has the smallest initial code size, but also required the most changes to be resilient. Overall, the proposed countermeasures, both generic and specific, make the reference implementations resilient to fault attacks with minimal impact on performance and size.

7 Conclusion

In this paper we present the first fault attack that directly targets the WOTS schema, which is an integral part of all currently standardised HBS schemes. Therefore, it affects LMS, XMSS and SPHINCS$^+$. Furthermore, our attack affects both signature generation and signature verification. Although the attack requires brute-force computation of an appropriate digest, we have demonstrated its feasibility. Our research shows that for a Winternitz parameter $w = 16$, signatures are forgeable for all algorithms with a NIST security level up to 3. For $w = 256$, signatures generated by all algorithms considered are forgeable, regardless of the chosen security level. The complexity of the attack is at most affected by the choice of the Winternitz parameter and the internal tree structure. Choosing a larger value for w combined with a multi-tree structure leads to parameter sets that can be broken within seconds with a single GPU. To defend against this attack, appropriate countermeasures must be in place. The analysis of the proposed countermeasures shows their effectiveness and efficiency against the WOTS-specific attack. Furthermore, our proposed generic countermeasures harden the implementations so that a fault attack is no longer feasible within this scenario. However, despite the advanced progress in standardisation, our research has shown that the analysis of the implementation security of HBS algorithms is still an ongoing task. With our work, we aim to stimulate further research in this area. The recent selection of SPHINCS$^+$ for standardisation makes this particularly important, as this will lead to more vendors looking to incorporate HBS schemes into their products. We also see a need for a more thorough analysis of implementation security in general. A combination of algorithmically formalised knowledge and automated analysis could ensure a higher probability of early detection of vulnerabilities in implementations. Efforts in this direction will allow the development of hardened PQC implementations in a secure and rapid manner.

Acknowledgements This work was partly funded by the German Federal Ministry of Education and Research (BMBF) in the project APRIORI under grant number 16KIS1390.

Bibliography

[AdGHB] Ever Atilano, Arnaud de Grandmaison, Karine Heydemann, and Guillaume Bouffard. Assessing the effectiveness of MCUboot protections against fault injection attacks.

[ALCZ20] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. FPGA-based SPHINCS+ Implementations: Mind the Glitch. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 229–237, 2020.

[ANS22] ANSSI. ANSSI views on the Post-Quantum Cryptography transition. `https://www.ssi.gouv.fr/en/publication/anssi-views-on-the-post-quantum-cryptography-transition/`, January 2022.

[Aum19] Jean-Philippe Aumasson. Too Much Crypto. `https://eprint.iacr.org/2019/1492.pdf`, 2019.

[Ban] Tamas Ban. HW Fault Injection Mitigation - Trusted Firmware M. `https://www.trustedfirmware.org/docs/TF-M_fault_injection_mitigation.pdf`.

[BDE+11] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the Security of the Winternitz One-Time Signature Scheme. Cryptology ePrint Archive, Paper 2011/191, 2011. `https://eprint.iacr.org/2011/191`.

[BDK+] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle Signatures with Virtually Unlimited Signature Capacity. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, volume 4521, pages 31–45. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):199–224, Feb. 2019.

[BHRVV] Joppe W. Bos, Andreas Hülsing, Joost Renes, and Christine Van Vredendaal. Rapidly Verifiable XMSS Signatures. pages 137–168.

[Bit22] Bitmain Antminer S19 XP (140Th) profitability. https://www.asicminervalue.com/miners/bitmain/antminer-s19-xp-140th, July 2022.

[Bro] David Brown. Post-quantum cryptography. `https://github.com/mcu-tools/mcuboot/discussions/1099?sort=top`.

[BSI22] BSI. BSI – Technische Richtlinie: Kryptographische Verfahren: Empfehlungen und Schluessellaengen. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile`, January 2022.

[CAD+] David A. Cooper, Daniel C. Apon, Quynh H. Dang, Michael S. Davidson, Morris J. Dworkin, and Carl A. Miller. Recommendation for Stateful Hash-Based Signature Schemes.

[Cis19] Cisco. Post quantum trust anchors. `https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/docs/post-quantum-trust-anchors-wp.pdf`, 2019.

[CMP18] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. Grafting Trees: A Fault Attack Against the SPHINCS Framework. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, volume 10786, pages 165–184. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[Cro20] Sam Croley. Hashcat v6.1.1 benchmark on the Nvidia RTX 3090. `https://gist.github.com/Chick3nman/32e662a5bb63bc4f51b847bb422222fd`, September 2020.

[Cro22] Sam Croley. Hashcat v6.2.6 benchmark on the Nvidia RTX 4090. `https://gist.github.com/Chick3nman/32e662a5bb63bc4f51b847bb422222fd`, October 2022.

[FKK+22] Michael Fahr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, and Daniel Apon. When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 979–993, Los Angeles CA USA, November 2022. ACM.

[GBH18] Leon Groot Bruinderink and Andreas Hülsing. "Oops, I Did It Again" – Security of One-Time Signatures Under Two-Message Attacks. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, volume 10719, pages 299–322. Springer International Publishing, Cham, 2018.

[gen20] IT Security Solutions From genua Withstand Attacks With Quantum Computers. `https://www.genua.eu/knowledge-base/it-security-solutions-from-genua-withstand-attacks-with-quantum-computers`, 2020.

[Gen23] Aymeric Genêt. On Protecting SPHINCS+ Against Fault Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 80–114, March 2023.

[GKPM18] Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical Fault Injection Attacks on SPHINCS. 2018. `https://eprint.iacr.org/2018/674`.

[goo] `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/LUczQNCw7HA/m/f5OWvA3RBAAJ`.

[GTSC] James Gratchoff, Niek Timmers, Albert Spruyt, and Lukasz Chmielewski. Proving the wild jungle jump. *Technical report, University of Amsterdam, Tech. Rep.*

[HBD+20] A. Hülsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kolbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, , and W. Beullens. SPHINCS+ - submission to the NIST post-quantum project, v.3. `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`, 2020.

[HBG$^+$18] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. XMSS: eXtended Merkle Signature Scheme. `https://datatracker.ietf.org/doc/html/rfc8391`, May 2018.

[HGA$^+$21] Florian Hauschild, Kathrin Garb, Lukas Auer, Bodo Selmke, and Johannes Obermaier. ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults. In *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 20–30, September 2021.

[HK22] Andreas Hülsing and Mikhail Kudinov. Recovering the tight security proof of $SPHINCS^+$. Cryptology ePrint Archive, Paper 2022/346, 2022. `https://eprint.iacr.org/2022/346`.

[HMU$^+$20] Johann Heyszl, Katja Miller, Florian Unterstein, Marc Schink, Alexander Wagner, Horst Gieser, Sven Freud, Tobias Damm, Dominik Klein, and Dennis Kügler. Investigating Profiled Side-Channel Attacks Against the DES Key Schedule. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 22–72, June 2020.

[Hü] Andreas Hülsing. W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918, pages 173–188. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[KGC$^+$20] Vinay B. Y. Kumar, Naina Gupta, Anupam Chattopadhyay, Michael Kasper, Christoph Krauß, and Ruben Niederhagen. Post-quantum secure boot. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1582–1585, 2020.

[KHRY22] Mikhail Kudinov, Andreas Hülsing, Eyal Ronen, and Eylon Yogev. SPHINCS+C: Compressing SPHINCS+ With (Almost) No Cost. Cryptology ePrint Archive, Paper 2022/778, 2022. `https://eprint.iacr.org/2022/778`.

[KPC$^+$] Panos Kampanakis, Peter Panburana, Michael Curcio, Chirag Shroff, and Mahbub Alam. Post-quantum LMS and SPHINCS+ hash-based signatures for UEFI secure boot. page 22.

[MAA$^+$] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith-Tone, and Jacob Alperin-Sheriff. Status report on the third round of the NIST post-quantum cryptography standardization process.

[mcu] MCUboot documentation. `https://docs.mcuboot.com/`.

[Mer90] Ralph C. Merkle. A Certified Digital Signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, volume 435, pages 218–

238. Springer New York, New York, NY, 1990. Series Title: Lecture Notes in Computer Science.

[MMC19] D. McGrew and S. Fluhrer M. Curcio. Leighton-Micali Hash-Based Signatures. `https://datatracker.ietf.org/doc/html/rfc8554`, April 2019.

[O'F19] Colin O'Flynn. MIN()imum failure: EMFI attacks against USB stacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

[OSS17] Johannes Obermaier, Robert Specht, and Georg Sigl. Fuzzy-glitch: A practical ring oscillator based clock glitch attack. *2017 International Conference on Applied Electronics (AE)*, pages 1–6, 2017.

[Phi22] Jade Philipoom. Request for feedback on possible SPHINCS+ variant. `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/LUczQNCw7HA/m/f50WvA3RBAAJ`, December 2022.

[Rai22] Guillaume Raimbault. Welcome to a new generation of future-proof TPMs: OPTIGA TPM SLB 9672. `https://www.infineon.com/dgdl/Infineon-OPTIGA-TPM-SLB9672.pdf?fileId=8ac78c8b7e7122d1017f071c3f6b00d2`, February 2022.

[Rot19] Thomas Roth. TrustZone-M(eh): Breaking ARMv8-M's security, 2019.

[Son19] Sondero. Hashcat v5.1.0 benchmark on the Intel(R) Core(TM) i7-9700K. `https://hashcat.net/forum/thread-9042-post-47927.html#pid47927`, December 2019.

[SZK$^+$18] Bodo Selmke, Kilian Zinnecker, Philipp Koppermann, Katja Miller, Johann Heyszl, and Georg Sigl. Locked out by Latch-up? An Empirical Study on Laser Fault Injection into Arm Cortex-M Processors. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 7–14, Amsterdam, Netherlands, September 2018. IEEE.

[VCGS13] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Security Evaluations beyond Computing Power: How to Analyze Side-Channel Attacks You Cannot Mount? In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Thomas Johansson, and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881, pages 126–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. Series Title: Lecture Notes in Computer Science.

[WJW$^+$] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems: XMSS hardware accelerators for RISC-v. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography*

– *SAC 2019*, volume 11959, pages 523–550. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[WOS22] Alexander Wagner, Felix Oberhansl, and Marc Schink. To Be, or Not to Be Stateful: Post-Quantum Secure Boot Using Hash-Based Signatures. In *Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security*, ASHES'22, page 85–94, New York, NY, USA, 2022. Association for Computing Machinery.