Practical Fault Injection Attacks on SPHINCS

Aymeric Genêt[1,2], Matthias J. Kannwischer[3] , Hervé Pelletier[2], and Andrew McLauchlan[2] ⋆

[1] EPFL, Lausanne, Switzerland
aymeric.genet@epfl.ch
[2] Kudelski Group, Cheseaux-sur-Lausanne, Switzerland
{herve.pelletier,andrew.mclauchlan}@nagra.com
[3] Digital Security Group, Radboud University, Nijmegen, The Netherlands
matthias@kannwischer.eu

Abstract. The majority of currently deployed cryptographic public-key schemes are at risk of becoming insecure once large scale quantum computers become practical. Therefore, substitutes resistant to quantum attacks—known as post-quantum cryptography—are required. In particular, hash-based signature schemes appear to be the most conservative choice for post-quantum digital signatures.

In this work, we mount the first practical fault attack against hash-based cryptography. The attack was originally proposed by Castelnovi et al. [8] and allows the creation of a universal signature forgery that applies to all current standardisation candidates (XMSS, LMS, SPHINCS$^+$, and Gravity-SPHINCS). We perform the attack on an Arduino Due board featuring an ARM Cortex-M3 microprocessor running the original stateless scheme SPHINCS with a focus on practicality. We describe how the attack is mountable with a simple voltage glitch injection on the targeted platform, which allowed us to collect enough faulty signatures to create a universal forgery within seconds.

As the attack also applies to stateful schemes, we show how caching one-time signatures can entirely prevent the attack for stateful schemes, such as XMSS and LMS. However, we discuss how protecting stateless schemes, like SPHINCS, SPHINCS$^+$, and Gravity-SPHINCS, is more challenging, as this countermeasure does not apply as efficiently as in stateful schemes.

Keywords: SPHINCS · hash-based signature · voltage glitching · fault attack · digital signature

1 Introduction

With the arrival of large-scale quantum computers, current public-key cryptography is at risk. This is the result of Shor's quantum algorithm [18] which is able to efficiently factor large integers and solve the discrete logarithm problem. As a

⋆ This work was done while Matthias J. Kannwischer was at TU Darmstadt and the University of Surrey.

result, substitutes resistant to attacks by quantum computers—also known as *post-quantum cryptosystems*—need to be developed.

One family of post-quantum cryptography known for creating digital signatures with the sole use of a cryptographic hash function is *hash-based cryptography*. The main advantage of such schemes is that their security relies only on certain cryptographic properties of a generic hash function. Thus, if the chosen hash functions are broken in the future, they can be easily replaced with new hash constructions. Besides, from a quantum standpoint, the effectiveness of generic attacks against hash functions cannot exceed Grover's algorithm [19], i.e., a quantum adversary cannot obtain more than a square-root speed-up compared to classic preimage search.

The urge to transition from current cryptosystems to post-quantum cryptography has recently gained a lot of popularity [17], mainly due to progress in quantum computing. In 2017, 82 proposals for post-quantum cryptography were received by NIST of which 69 submissions were accepted for a first round of evaluation.

While many researchers contributing to NIST's evaluation focus on theoretical cryptanalysis, implementation aspects also need to be assessed and are expected to be an important criterion for the selection of future post-quantum cryptography standards. Physical attacks have been shown to have disastrous impact on certain implementations of cryptography, often completely compromising the security at low cost. Indeed, while post-quantum schemes need to resist quantum computers, they must run on classical computers, such as embedded devices. This paper evaluates the susceptibility of hash-based signatures—in particular, SPHINCS [3]—to fault injection attacks.

Fault injection attacks. A *fault*, either natural or malicious, is a misbehaviour of a device that causes the computation to deviate from its specification. For example, a fault can flip bits in a certain memory cell, corrupting the value held in this register. In a fault injection attack, an adversary is able to actively inject malicious faults into a cryptographic device, such that its process outputs faulty data. Invalid outputs, potentially combined with valid ones, are then used to reconstruct parts of secret data, such as signing keys.

Research during the last two decades found that many widely used schemes, when implemented without specific fault protection, can be broken by fault attacks. The first successful attack dates back to 1997 [4], where Boneh et al. exploited both a faulty and a valid RSA signature to recover the private key of a device.

These attacks are primarily relevant for embedded cryptographic devices like smart cards, where it is reasonable to assume that an adversary has direct access to the device and can control when cryptographic operations are performed.

Faults can be induced in various ways. The most classical ones include exposing the device to voltage variations or manipulating the clock frequency, leading to operation outside of the tolerance of the cryptographic devices.

When analysing the fault vulnerability of an implementation, the capability of the adversary needs to be taken into account. This involves measuring the

accuracy of the injected faults in terms of location, timing, number of bits affected, success probability, duration, ... The more precisely the fault injection can be tuned, the more powerful are the possible attacks.

Related work. Mozaffari-Kermani et al. [15] propose and evaluate the performance of countermeasures that protect Merkle tree constructions from natural and malicious faults. However, they neither evaluate the practicality of fault injection attacks against unprotected implementations, or the effectiveness of their countermeasure. Work by Castelnovi et al. [8], performed partially in parallel to ours, proposes the same attack as we do, but does not provide a practical verification. Kannwischer et al. [14] recently analysed the differential power analysis vulnerability of XMSS and SPHINCS. They limit their analysis to purely passive adversaries, i.e., entirely exclude fault injection attacks.

Contributions. This work shows the first practical fault attack on hash-based signatures with a focus on SPHINCS [3]—the stateless hash-based signature scheme that led to SPHINCS$^+$ [11] and Gravity-SPHINCS [2], both candidates to the NIST standardisation process. The attack is based on the existing work of Castelnovi et al. [8] who analysed the impact of a fault injection on the security of SPHINCS-like structures, but without proof of concept. This paper brings new insights into the subject by focusing on the practical details to perform the fault attack on a generic embedded device. We show how a low-cost injection of a single glitch is enough capability to obtain exploitable faulty signatures, and highlight which procedures are critical to protect. Finally, we conclude on a discussion on how to completely thwart the attack on stateful schemes, and why protecting stateless schemes is a difficult task.

2 Preliminaries

Hash-based signature schemes use hash functions to create digital signatures. They mainly provide constructions for one-time signatures (OTS) and few-time signatures (FTS), which can then be combined with a binary hash tree (Merkle tree) to design many-time signatures (MTS). As their names indicate, an OTS must not be used more than once, the security of an FTS degrades over uses, and an MTS can be used only a specific number of times.

Until recently, practical hash-based signatures schemes were *stateful*, meaning that a state needed to be maintained as a part of the secret key (e.g., XMSS [6,12], and LMS [16]). This introduced a severe requirement for the executing device and prevented a drop-in replacement of schemes currently deployed. In 2015, Bernstein et al. proposed SPHINCS [3], the first practical scheme that has the particularity of being *stateless*. This was a huge step in the development of hash-based schemes, as they can potentially become the next digital signatures standard in a post-quantum world.

Recently, two extensions of SPHINCS have been submitted to the NIST post-quantum standardisation project [17]: SPHINCS$^+$ [11] and Gravity-SPHINCS [2]. Despite such improvements, this paper focuses on the original SPHINCS scheme.

However, due to their similarity to SPHINCS, SPHINCS$^+$ and Gravity-SPHINCS can be attacked in a very similar way.

Due to space limitations, we limit the preliminaries to the parts of the schemes relevant for the fault injection attack. Particularly, we skip the signature verification of the schemes, as well as the FTS scheme HORST since this building block is not targeted. For self-contained descriptions of the algorithms, we refer the reader to the original publications [3, 6, 10] or an introduction to hash-based signatures.

2.1 W-OTS$^+$

W-OTS$^+$, as well as its predecessor the Winternitz one-time signature (W-OTS), implements a time/space trade-off parametrised by the Winternitz parameter $w = 2^W$. A small value of w results in a faster scheme, but leads to larger keys and signatures, as the scheme signs W bits at once by hashing a secret value at most $2^W - 1$ times.

For a security parameter $n \in \mathbb{N}$, given a second-preimage resistant hash function $F : \{0,1\}^n \to \{0,1\}^n$ and a set of bitmasks $\mathbf{r} = (r_1, \ldots, r_{w-1}) \in \{0,1\}^{(w-1)\times n}$, the W-OTS$^+$ chaining function $c^i(x, \mathbf{r})$ is defined as

$$\begin{cases} c^0(x, \mathbf{r}) = x \\ c^i(x, \mathbf{r}) = F(c^{i-1}(x, \mathbf{r}) \oplus r_i),\ i < w. \end{cases}$$

Also, let the following lengths be

$$\ell_1 = \left\lceil \frac{n}{W} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{W} \right\rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

W-OTS$^+$.keyGen Given security parameter n,

1. Choose secret seed $\mathcal{S} \in \{0,1\}^n$ and randomisation bitmasks $\mathbf{r} \in \{0,1\}^{n\times(w-1)}$.
2. Expand $\mathcal{S}$ to $\mathrm{sk} = (\mathrm{sk}_1, \ldots, \mathrm{sk}_\ell) \in \{0,1\}^{\ell\times n}$.
3. Compute public key $\mathrm{pk} = (\mathrm{pk}_1, \ldots, \mathrm{pk}_\ell) = \left(c^{w-1}(\mathrm{sk}_1, \mathbf{r}), \ldots, c^{w-1}(\mathrm{sk}_\ell, \mathbf{r})\right)$.
4. Output:
   - Secret key: $\mathcal{S}$.
   - Public key: randomisation bitmasks Q and pk.

W-OTS$^+$.sign Given message $M \in \{0,1\}^n$, secret seed $\mathcal{S}$ and bitmasks $\mathbf{r}$,

1. Split M into blocks of W bits $M = (b_1, \ldots, b_{\ell_1})$ with $b_i \in \{0, \ldots w-1\}$.
2. Compute checksum C from M and split up into blocks of W bits.
3. Concatenate M and C such that $B = M||C = (b_1, \ldots, b_\ell)$.
4. Re-compute sk from $\mathcal{S}$.
5. Compute signature as $\sigma_{\text{W-OTS}^+} = (\sigma_1, \ldots, \sigma_\ell) = \left(c^{b_1}(\mathrm{sk}_1, \mathbf{r}), \ldots, c^{b_\ell}(\mathrm{sk}_\ell, \mathbf{r})\right)$.

W-OTS$^+$.keyExtract To compute the pk from $\sigma_{\text{W-OTS}^+}$ and M,

1. Re-compute $(b_1, \ldots, b_\ell)$ from $B = M||C$ as above.
2. Compute $\mathrm{pk}_i = c^{w-1-b_i}(\sigma_i, \mathbf{r})$ for $1 \le i \le \ell$.

2.2 MSS

SPHINCS uses the Merkle Signature Scheme (MSS) to construct an MTS that combines many W-OTS$^+$ key pairs using a Merkle tree. Given a security parameter n, the scheme requires a second-preimage resistant hash function $H : \{0,1\}^{2n} \rightarrow \{0,1\}^n$.

MSS.keyGen Given Merkle tree height h,

1. Choose secret seed $\mathcal{S}_{\text{MSS}} \in \{0,1\}^n$ and randomisation bitmasks $Q \in \{0,1\}^{2n\times h}$.
2. Expand $\mathcal{S}_{\text{MSS}}$ to $\mathcal{S}_i$ for $0 \le i < 2^h$.
3. Generate 2^h W-OTS$^+$ key pairs using $\mathcal{S}_i$ for $0 \le i < 2^h$.
4. Organise W-OTS$^+$ public keys as a binary hash tree, where
   (a) Leaves $v_0[j]$ $(0 \le j < 2^h)$ consist of the compressed public keys (see [12])
   (b) Inner nodes $v_i[j]$ are computed as $v_i[j] = H((v_{i-1}[2j]||v_{i-1}[2j+1]) \oplus Q_i)$ for $1 \le i \le h$ and $0 \le j < 2^{h-i}$.
5. Output:
   – Secret key: $\mathcal{S}_{\text{MSS}}$ and index of next unused leaf $s := 0$.
   – Public key: randomisation bitmasks Q and root of the Merkle tree $v_h[0]$.

MSS.sign Given message $M \in \{0,1\}^n$, $\mathcal{S}_{\text{MSS}}$, s, and Q

1. Re-compute the W-OTS$^+$ secret key $\mathcal{S}_s$ from $\mathcal{S}_{\text{MSS}}$.
2. Create $\sigma_{\text{W-OTS}^+}$ for M using $\mathcal{S}_s$.
3. Increment next unused leaf counter s.
4. Compute the authentication path for the selected W-OTS$^+$ key pair. The authentication path consists of the nodes in the MSS tree required to recompute its root:

$$\text{Auth} = \left(v_0[s \oplus 1], \ldots, v_{h-1}\left[\lfloor s/2^{h-1} \rfloor \oplus 1\right]\right)$$

5. Output $\sigma_{\text{W-OTS}^+}$, index s, and the authentication path.

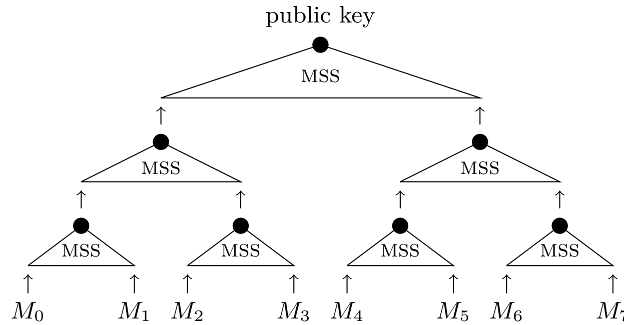

Fig. 1. Example of an instance of the tree chaining method CMSS. The hypertree is of height $h = 3$ and consists of $d = 3$ layers of sub trees.

2.3 CMSS

While MSS allows the signing of a large number of messages by choosing a large h, there are still some limitations: the key generation and signing require $\mathcal{O}(2^h)$ operations. While the former can be solved using optimised authentication path computation, the latter is addressed by CMSS [7]; an improved hash-based scheme that chains multiple MSS together. CMSS uses a multi-layered hypertree of MSS where the upper layers sign the root of the lower layers. The lowest layer is used to sign messages.

CMSS.keyGen Given total height h and number of layers d,

1. Choose $\mathcal{S}_{\text{CMSS}} \in \{0,1\}^n$ which is used to derive $\mathcal{S}_{\text{MSS},i,j}$ for each subtree.
2. Choose randomisation bitmasks Q for MSS and W-OTS$^+$
3. Generate top-most tree of height h/d using $\mathcal{S}_{\text{MSS},d-1,0}$.
4. Output:
   - Secret key: $\mathcal{S}_{\text{CMSS}}$ and index of next unused leaf $s := 0$.
   - Public key: root of the topmost tree and randomisation bitmasks Q.

Fig. 1 illustrates a CMSS hypertree for $d = 3$ and $h = 3$, resulting in intermediate Merkle trees of height $h/d = 1$ with 2 leaves.

CMSS.sign Given a message $M \in \{0,1\}^n$, $\mathcal{S}_{\text{CMSS}}$, and Q

1. Expand $\mathcal{S}_{\text{CMSS}}$ to the corresponding seeds $\mathcal{S}_{\text{MSS},i,j}$ for each $0 \leq i < h/d$ where $j = \left\lfloor s/(2^{(i+1)(h/d)}\right\rfloor$.
2. Create a MSS signature for M using the bottom tree corresponding to s.
3. Increment next unused leaf counter s.
4. Create one MSS signature for each non-bottom layer, where the Merkle trees on the upper layers are used to sign the roots of the lower layers.

2.4 SPHINCS

The major downside of stateful hash-based signature schemes like CMSS is that each leaf of the hypertree must not be used more than once which requires securely maintaining a state.

If an adversary is able to manipulate the state, such that the signer uses the same key pair multiple times, the security of CMSS and other stateful hash-based signature schemes degrades significantly. This, among other reasons, prevents the wide adoption of stateful hash-based signature schemes.

To address the situation, the SPHINCS proposal [3] has made the elimination of the state possible by bringing two alterations to a huge CMSS. First, the OTS on the lowest layer of the hypertree are replaced with the FTS scheme HORST [4]. Second, the leaves are not used consecutively, but are selected pseudorandomly, depending on the message and the secret key. Fig. 2 illustrates a small example of a SPHINCS hypertree. Note that all layers, except for the bottom one, still work as in CMSS.

[4] For details on the HORST signing procedure, we refer to [3].

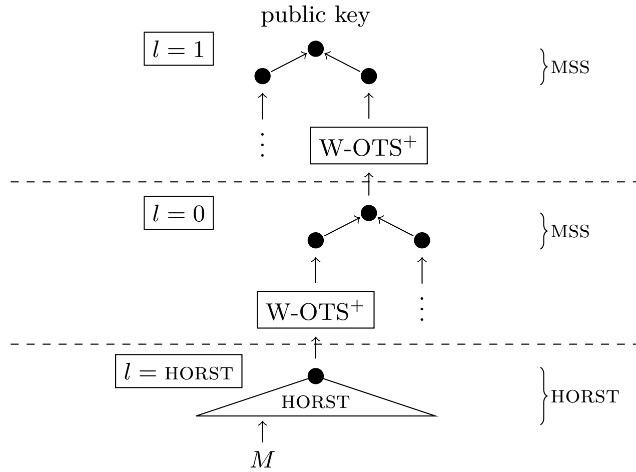

Fig. 2. Example of SPHINCS.

SPHINCS.keyGen

1. Choose secret key $\mathrm{SK} = (\mathrm{SK}_1, \mathrm{SK}_2) \in \{0,1\}^n \times \{0,1\}^n$ uniformly at random.
   - SK_1 is used for pseudorandom generation of W-OTS$^+$ and HORST keys.
   - SK_2 is used for pseudorandom selection of a HORST instance for signing.
2. Choose random bitmasks Q for MSS, W-OTS$^+$, and HORST
3. Compute single MSS tree on the top layer.
4. Output:
   - Secret key: SK.
   - Public key: randomisaton bit masks Q and root of top MSS tree.

SPHINCS.sign Given message M, secret key SK, and Q

1. Compute pseudorandom values R_1, R_2 from M and SK_2.
2. Compute randomised message digest D from M and R_1.
3. Re-compute HORST key pair at index R_2.
4. Create HORST signature σ_H for D.
5. Compute CMSS signature to authenticate the HORST public key.
6. Output signature consisting of:
   - R_1 and R_2.
   - The HORST signature σ_H.
   - The CMSS signature consisting of d MSS signatures.

3 Fault attack on SPHINCS

This section describes how a universal SPHINCS signature forgery can be created by injecting faults during its signing procedure. First, we explain the different steps of the attack and highlight the procedures that lead to exploitable faulty signatures. Then, the described attack is simulated to illustrate the complexity of the attack in terms of faulty signatures obtained. The section ends with a practical verification on an Arduino Due board using voltage glitching.

3.1 Attack description

The attack is based upon Castelnovi et al. [8] who showed in 2018 that SPHINCS-like schemes are notably intolerant to the presence of faults during execution. This is because their CMSS structure uses OTS to sign the roots of intermediate Merkle trees. Since the subtrees are not supposed to change from one execution to another, they are re-computed on-the-fly and re-signed using the same OTS instance. Therefore, a corruption on the root computation results in an OTS instance signing a different message, weakening its security.

Attacking SPHINCS with faults first requires signing a message M to obtain a valid signature

$$\Sigma = (R_1, R_2, \sigma_H, \sigma_{W,0}, \mathrm{Auth}_0, \ldots, \sigma_{W,d-1}, \mathrm{Auth}_{d-1}).$$

By re-signing the same M, the scheme will produce the same signature, passing through the exact same path of the hypertree. However, if an error occurs during the construction of any *non-top* subtree $0 \le i < d-1$, the algorithm will output a different signature $\widehat{\Sigma}$ where $\widehat{\sigma}_{W,i} \neq \sigma_{W,i}$.

Combining the secret values from $\widehat{\sigma}_{W,i}$ and $\sigma_{W,i}$, the attack consists of attempting to forge a signature $\sigma'_{W,i}$ for a known subtree. Once the subtree is successfully forged, it can be used to maliciously produce $(R'_1, R'_2, \sigma'_H, \sigma'_{W,j}, \mathrm{Auth}_j)$ for $0 \le j \le i$ and, thus, maliciously signs an arbitrary M'. The rest of the signature is simply taken from the valid signature, i.e., $(\sigma'_{W,j}, \mathrm{Auth}'_j) = (\sigma_{W,j}, \mathrm{Auth}_j)$ for $i < j < d$.

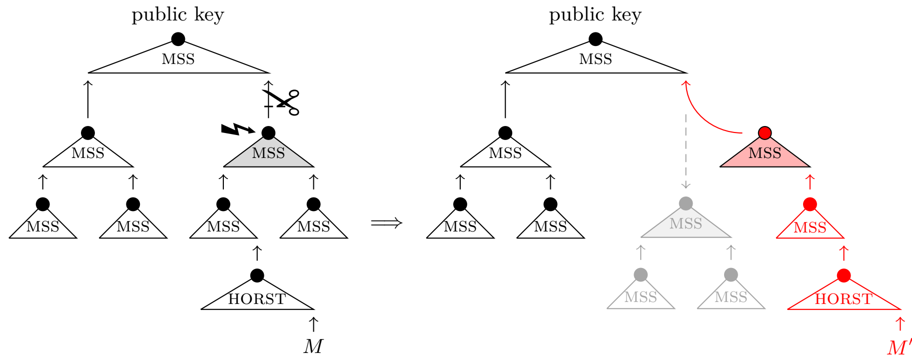

Fig. 3. An illustration of the tree grafting against SPHINCS.

An illustration of the attack is shown in Fig. 3. The picture on the left shows a message M being signed with a SPHINCS hypertree while the highlighted subtree is being attacked. Then, on the right, the hypertree is cut at this subtree and *grafted* with a branch of forged subtrees, allowing an arbitrary message M' to be signed.

Processing faulty signatures. As the fault attack forces an OTS to be reused, a post-processing procedure is required to exploit the faulty signatures. Since we have no information on the corrupted messages, the W-OTS$^+$ secret values inside the faulty signatures need to be identified. This is done by using the W-OTS$^+$ public key, which can be extracted only with a correct W-OTS$^+$ signature of the attacked subtree.

Using the public key of the W-OTS$^+$ instance, the secret values can be identified by correctly guessing all the blocks of the corrupted subtree root. Each block can be confirmed separately by applying $c^i(x, \mathbf{r})$ a number of times equivalent to the presumed value of the block. The resulting values are stored in σ'_i and correspond to the blocks b'_i for $0 \leq i < \ell$.

Once the W-OTS$^+$ secret values have been recognized, the universal forgery is created by trying to graft a subtree on the existing SPHINCS structure. For this, the attack creates a subtree from a random secret key SK'_1 and tries to sign its root using the recovered values. If $(b_1, \ldots, b_\ell)$ denote the result of splitting the root in W-OTS$^+$ blocks, the signature succeeds if $b_i \leq b'_i$ for $1 \leq i \leq \ell$. In this case, a valid W-OTS$^+$ signature for the grafted subtree is then forged. This subtree can then be used to sign the root of all the previous subtrees, which consequently allows universal forgeries. Otherwise, the attempt can be repeated over with a different SK'_1 until it succeeds.

Following the above processing steps, we derived our own post-processing algorithm in Fig. 4. This algorithm optimally exploits the information obtained from the available faulty signatures to create a universal forgery on SPHINCS. An implementation in Python of the algorithm is made available at [9].

Attack complexity. In order to succeed, the adversary needs to find a Merkle tree that can be correctly signed using the secret values from q different faulty signatures. This corresponds to drawing random b_i for such that all $b_i \leq b'_i$ for $1 \leq i \leq \ell$.

To measure the number of attempts to obtain $b_i \leq b'_i$ for $1 \leq i \leq \ell$, we approximate the probability of such an occurrence with a generalization of the work of Bruinderink et al. [5]. Assuming that all blocks $b_1, \ldots, b_\ell$ are uniformly random, we obtain:

$$\mathbb{P} = \frac{1}{w^\ell} \left(\sum_{x=0}^{w-1} \left(1 - \left(\frac{w-(x+1)}{w} \right)^{q+1} \right) \right)^\ell . \tag{1}$$

For example, as $w = 16$ and $\ell = 67$ for SPHINCS, if an adversary obtains $q = 20$ faulty signatures, the W-OTS$^+$ forgery succeeds with a probability of $\mathbb{P} \approx 0.243$. Since the second part of the algorithm follows a geometric distribution, the expected number of attempts is simply $1/0.243 \approx 4.12$.

Note that for the last ℓ_2 blocks (i.e., the checksum), the assumption of uniformity does not hold. This is because the blocks actually correspond to a sum of uniform variables. This leads to a slightly lower success probability in practice, which will be highlighted in our simulations.

SPHINCS$^+$ and Gravity-SPHINCS. As noticed by Castelnovi et al. [8], the attack also applies to the newer schemes SPHINCS$^+$ and Gravity-SPHINCS.

Require: M : the valid subtree root
Require: σ_W : the valid W-OTS$^+$ signature for M
Require: $\widehat{\sigma}_{W,i}$: q faulty signatures
1: pk $\leftarrow$ **W-OTS$^+$.keyExtract**(M, σ_W)
2: Initialise $(\theta_1, \ldots, \theta_\ell)$ with $(\sigma_1, \ldots, \sigma_\ell)$ from σ_W
3: Initialise $(b_1, \ldots, b_\ell)$ with M and its checksum C
4: **for each** $\widehat{\sigma}_{W,i}$ **do**
5:     **for each** $\widehat{\sigma}_j$ **in** $\widehat{\sigma}_{W,i} = (\widehat{\sigma}_1, \ldots, \widehat{\sigma}_\ell)$ **do**
6:         Identify $\widehat{b}_j$ s.t. $c^{w-1-\widehat{b}_j}(\widehat{\sigma}_j, \mathbf{r}) = \text{pk}_j$
7:         **if** $\widehat{b}_j < b_j$ **then**
8:             $b_j \leftarrow \widehat{b}_j$ and $\theta_j \leftarrow \widehat{\sigma}_j$
9:         **end if**
10:     **end for**
11: **end for**

12: Initialise $\sigma' = (\sigma'_1, \ldots, \sigma'_\ell)$
13: Draw $\text{SK}'_1 \in \{0,1\}^n$ at random
14: Create Merkle tree of root M' with SK'_1
15: Initialise $(b'_1, \ldots, b'_\ell)$ with M' and its checksum C'
16: **for each** b'_i **in** $(b'_1, \ldots, b'_\ell)$ **do**
17:     **if** $b_i \leq b'_i$ **then**
18:         $\sigma'_i \leftarrow c^{b'_i - b_i}(\theta_i, \mathbf{r})$
19:     **else**
20:         Go to line 13
21:     **end if**
22: **end for**
23: **return** (SK'_1, σ')

Fig. 4. Forgery procedure which finds an SK'_1 able to forge a signature for any message given a sufficient number of faulty signatures.

In Gravity-SPHINCS, the signing algorithm is still deterministic, which makes the attack almost equivalent to the one described above. The scheme however suggests caching the OTS at high levels, forcing the attacker to target subtrees at lower levels. This causes no problem, as the forged signature is grafted to the upper part of a valid one anyway. The authors have also replaced HORST with a safer FTS scheme; PORS, whose generation could also be attacked to obtain faulty signatures.

The SPHINCS$^+$ scheme, for its part, proposes an optional randomiser to avoid deterministic signing. Thanks to this feature, focusing a single subtree becomes harder, as even signing a same message takes a different path in the hypertree. However, as the subtrees on higher layers are likelier to repeat, the attacker might inject a fault on the penultimate layer (i.e., $i = d - 1$). More injections are required than with its deterministic variants, but because of the birthday paradox, a W-OTS$^+$ instance is being re-used within $2^{h/2d}$ successful corruption. Even with the maximum level of security, $h = 64$ and $d = 8$, making

a re-use successfully occur every 16 corruption in average. HORST in SPHINCS$^+$ is replaced by FORST, but is an inefficient target, as argued before.

3.2 Simulation

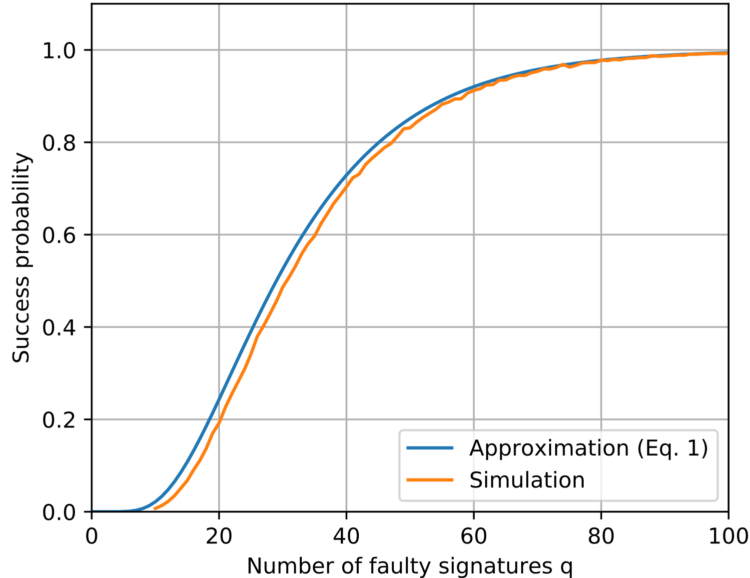

Fig. 5. Success rate of the W-OTS$^+$ forgery with $n = 256$ and $w = 16$ using a single forgery attempt. The simulated success probability is slightly lower than the approximation in Eq. 1 due to the checksum.

To evaluate the practicality of the attack and verify our expected complexity, we first simulated the fault injection attack. We modified the signature generation such that few bits were randomly flipped during the procedure. Then, we tried our forgery attack for different numbers of faulty signatures q.

The success probability that the forgery succeeds in a single attempt is illustrated in Fig 5. For comparison, the figure also contains the approximation in Eq. 1. The gap between the approximated and the simulated results is due to the inaccurate approximation of the checksum.

There exists an interesting trade-off on the number of faulty signatures and the complexity of the attack. This is because the partial intermediate hash chain values are sufficient to sign some messages. To evaluate this, we simulated the attack for certain values of q and the maximum number of forgery attempts p an adversary can afford. The results are shown in Fig. 6. For $q = 8$ an adversary still has a success probability of over 30% by attempting the forgery 64 times.

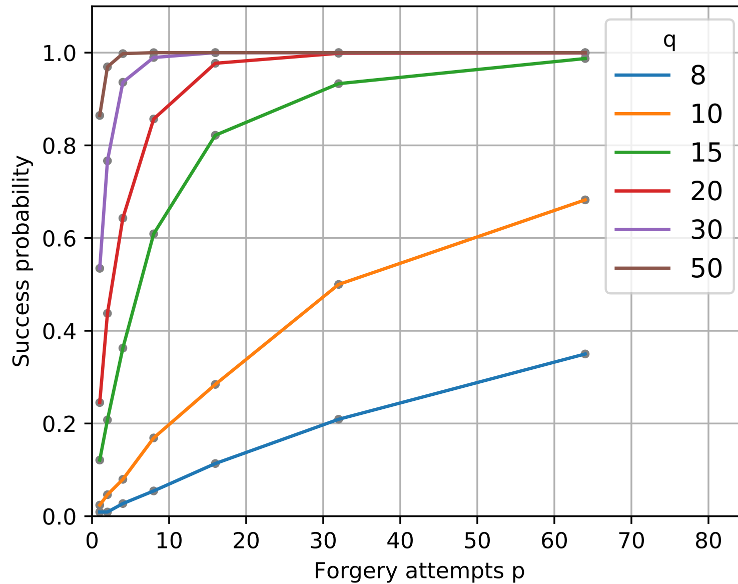

Fig. 6. Simulated success probability of the tree grafting attack given q faulty signatures. By investing more computational power (p) the adversary can forge signatures with fewer faulty signatures.

In practice, an adversary may be able to attempt many more forgeries, since the majority of the computation can be done offline and ahead of time. Even for $q = 1$, the forgery is expected to succeed after an average of 2^{34} different seeds are tried, which is still feasible within a reasonably long period of time [5].

3.3 Experimental verification on SPHINCS

The following subsection explains how the described fault attack can be performed on a real device. We chose to attack an Arduino Due board which features a SAM3X8E ARM Cortex-M3 microcontroller. A custom ARM implementation of SPHINCS-256, adapted from the ARMed SPHINCS implementation written by Hülsing et al. [13], was written for the project. Our code is made available at [9].

In our attack scenario, we fixed a secret key SK_1 and signed a message to first obtain a valid SPHINCS signature. Then, to speed up the process, we considered only a portion of the signing algorithm; namely, the construction of a single subtree. In other words, the attacked piece of code takes the address of a W-OTS$^+$ instance used in the path of the SPHINCS signature, constructs the subtree that contains this instance, and authenticates it by printing its authentication path within the subtree. The resulting root is then finally signed. Moreover, to obtain information on the duration and timing of this construction, a GPIO was toggled for the whole procedure.

To obtain faulty signatures, the code was provided with the address of the W-OTS$^+$ instance from the first subtree layer in the valid SPHINCS signature. We executed this subtree construction multiple times during which a single voltage glitch was injected in the core power domain. Note that, although these changes ease the experimentation, an adversary can mount the attack on the unmodified implementation with negligible additional effort.

Setup. The board was modified to permit the injection of transient faults during code execution with voltage glitching. The goal is to provide an isolated access to the power core (`VDDCORE`) where glitches will be injected. We refer to the official Atmel ARM-based SAM3X8E datasheet and schematics [1] to locate the elements that have been modified:

- The `VDDOUT` pin from the 144-lead LQFP package was lifted.
- The ferrite bead L5 MH2029-300Y was unsoldered, decoupling `VDDCORE` from `VDDPLL`.
- The end of the removed ferrite corresponding to `VDDCORE` was soldered to a wire.
- The other end (corresponding to `VDDPLL`) was wired to the lifted `VDDOUT` pin.
- Each capacitor on the path of `VDDOUT` was removed, i.e., C11 to C17.

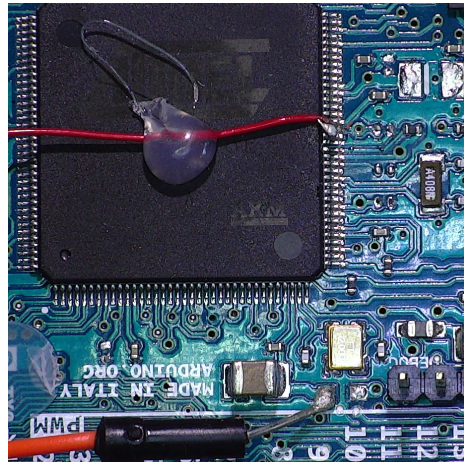

Fig. 7. Zoom on the modified part of the Arduino Due board.

With these modifications, a pulse generator is able to externally supply `VDDCORE` alone and inject glitches, while `VDDPLL` and `VDDOUT` were provided with a constant voltage. The resulting board can be seen in Fig. 7.

Fault process. Recording the GPIO trigger which was toggled during the first SPHINCS subtree construction indicates that the subtree construction takes

1026 ms. By synchronising on its rising edge, we would be able to target a specific subprocedure to inject glitch. The fault process requires a single corruption of *any* of the following subprocedures:

- W-OTS$^+$ public key generation at the leaves of a subtree, including:
  - Address computation.
  - Secret seed derivation with PRF.
  - Secret values derivation with PRNG.
  - Hash-chain application.
- L-tree compression of W-OTS$^+$ public key.
- Intermediate Merkle tree nodes computation.

The HORST layer can also be targeted, but the forgery of a HORST instance is more expensive than an MSS subtree. In our proof of concept, we arbitrarily toggled a second GPIO during the computation of the tree node $v_3[1]$ to synchronise our glitch injection. The injection was therefore controlled to randomly corrupt any instruction of this particular node compression.

Note that, given the length of the overall computation, glitching `VDDCORE` blindly by synchronising on the UART communication would have led to equivalent results. Also, the attack is still possible even if the adversary is unable to control the message but requires more faulty signatures. Moreover, if the fault occurs on a node below the authentication path, the resulting faulty signature is stealthy [8], meaning that a verifier will still accept it as valid.

The glitch injection is controlled with four parameters:

- The *delay* which controls the point in time at which the injection occurs.
- The *width* or the duration of the power outage.
- The initial *voltage* at which `VDDCORE` was supplied.
- The *depth* that describes how low the voltage drops.

These parameters were explored empirically until frequent corruptions of the node could be observed. The resulting values for each of them are given in Tab. 1.

Table 1. Parameters used to obtain faulty signatures.

Parameter	Value
Delay	195ns
Width	37ns
Voltage	0.85V
Depth	−3.1V

Using these parameters, we suspect that we hit the XOR operation between the hash function input bytes and the randomisation bitmasks. The glitch injection was performed 10000 times which led to a total of 85 different faulty signatures.

Results. Using the algorithm in Fig. 4 with the $q = 85$ faulty signatures, we were able to recover all of the attacked W-OTS$^+$ secret values. This obviously

allowed us to graft a subtree on the first trial. The forged subtree allowed us to forge another signature for a custom HORST instance which was then used to produce a signature for an arbitrary message. The signatures for the subtrees above these were simply taken from a valid signature, which resulted in an overall SPHINCS signature for a message of our choice. The details for all of these signatures are provided online in [9], as the results are too lengthy to be shown in paper.

To verify our simulation in Sec. 3.2, we also randomly picked $q = 20$ signatures from the obtained set of faulty signatures. The subtree forgery of Fig. 4 succeeded with an average of 4.6 attempts. This is close to the results simulated, which gives us confidence in the correctness of our assumptions. The least amount of signatures used to create a universal forgery is $q = 5$.

4 Countermeasures

Protecting hash-based schemes that implement CMSS, such as SPHINCS, from fault attacks is challenging. This is because the main vulnerability comes from the intermediate Merkle trees being signed by instances of OTS. As this feature is usually a core improvement in the practicality of these schemes, they cannot be "algorithmically fixed". Consequently, additional countermeasures need to be considered.

Currently, the only fault detection countermeasure for hash-based stateless digital signatures, such as SPHINCS, was developed by Mozaffari-Kermani et al. [15]. Their study first presents a method that detects faults by recomputing subtrees with swapped nodes, as well as an enhanced hash function that inherently protects against faults. This method unfortunately does not cover every aspect of the fault attack, as other vulnerable instructions need to be taken care of. Interestingly, their study was released before fault attacks on hash-based were researched.

The only way to completely thwart the fault threat in a CMSS structure is to compute every OTS *once*, and store them so the result can be output every time needed. For stateful schemes like XMSS^{MT} and the hierarchical variant of LMS, an efficient way of storing intermediate trees signatures is to consider a caching system. In this case, the cache only needs to store one OTS per layer of subtrees, and refresh them each time the signing algorithm discovers a new subtree. If a corruption occurs on a subtree while its signature is being cached, the adversary learns nothing about the secret key. However, the scheme would be disabled, as the cached OTS will always be wrong.

Similar caching techniques can also be designed for stateless schemes. Gravity-SPHINCS already recommends caching the top subtrees to speed up performance [2], which also happens to mitigate the attack. This technique is however not totally effective, as the attacker can still work around the cache to obtain faulty signatures.

Let us finally mention classic error detection and correction mechanisms. All the vulnerable instructions could be recomputed several times and compared to

each other, so a mismatch can be detected or the majority of the same result can be taken. These recomputations could be done by different hardware modules, so obtaining the same fault is unlikely. Note that faulty signatures still verify as valid signatures in the majority of cases. Thus, verifying after signing is not an effective way of detecting faults.

We consider innovative ways of protecting stateless hash-based signatures as interesting future work.

5 Conclusion

This work shows how SPHINCS-like structures on embedded devices can be easily defeated by low-cost fault injections. As the future NIST standards will definitely need to be industrialised, the candidates should be designed in a way that already protects against physical attacks. Unfortunately for SPHINCS and its variants, their designs are inherently vulnerable to fault attacks. Moreover, the lack of definitive countermeasure discourages embedded implementation.

Acknowledgments

We would like to thank Johannes Buchmann, Denis Butin, Juliane Krämer, Joost Rijneveld, and Peter Schwabe for helpful discussions and comments on earlier versions of this paper.

References

1. Atmel Corporation: SAM3X / SAM3A series datasheet, http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf. Accessed 2018-05-16.
2. Aumasson, J.P., Endignoux, G.: Gravity-SPHINCS. Tech. rep., National Institute of Standards and Technology (2017), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions
3. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: Practical Stateless Hash-Based Signatures. In: EUROCRYPT 2015. LNCS, vol. 9056, pp. 368–397. Springer (2015)
4. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: Fumy, W. (ed.) Advances in Cryptology - EUROCRYPT '97. Lecture Notes in Computer Science, vol. 1233, pp. 37–51. Springer (1997)
5. Bruinderink, L.G., 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 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10719, pp. 299–322. Springer (2017), https://doi.org/10.1007/978-3-319-72565-9_15

6. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS — A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In: Yang, B. (ed.) Post-Quantum Cryptography — 4th International Workshop, PQCrypto 2011. Lecture Notes in Computer Science, vol. 7071, pp. 117–129. Springer (2011)
7. Buchmann, J., García, L.C.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS — An Improved Merkle Signature Scheme. In: Barua, R., Lange, T. (eds.) Progress in Cryptology — INDOCRYPT 2006, 7th International Conference on Cryptology in India. Lecture Notes in Computer Science, vol. 4329, pp. 349–363. Springer (2006)
8. Castelnovi, L., Martinelli, A., Prest, T.: Grafting trees: A fault attack against the SPHINCS framework. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018. Lecture Notes in Computer Science, vol. 10786, pp. 165–184. Springer (2018)
9. Genêt, A., Kannwischer, M.J., Pelletier, H., McLauchlan, A.: Code used for the experimental verification of the SPHINCS fault attack, `https://github.com/sphincs-fi/sphincs-fi`
10. Hülsing, A.: W-OTS$^+$ — Shorter Signatures for Hash-Based Signature Schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) Progress in Cryptology — AFRICACRYPT 2013. Lecture Notes in Computer Science, vol. 7918, pp. 173–188. Springer (2013)
11. Hülsing, A., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P.: SPHINCS+. Tech. rep., National Institute of Standards and Technology (2017), available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`
12. Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. RFC 8391 (May 2018), `https://tools.ietf.org/html/rfc8391`
13. Hülsing, A., Rijneveld, J., Schwabe, P.: ARMed SPHINCS - computing a 41 KB signature in 16 KB of RAM. In: Cheng, C., Chung, K., Persiano, G., Yang, B. (eds.) Public-Key Cryptography - PKC 2016. Lecture Notes in Computer Science, vol. 9614, pp. 446–470. Springer (2016)
14. Kannwischer, M.J., Genêt, A., Butin, D., Krämer, J., Buchmann, J.: Differential power analysis of XMSS and SPHINCS. In: Fan, J., Gierlichs, B. (eds.) Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018. Lecture Notes in Computer Science, vol. 10815, pp. 168–188. Springer (2018)
15. Kermani, M.M., Azarderakhsh, R., Aghaie, A.: Fault Detection Architectures for Post-Quantum Cryptographic Stateless Hash-Based Secure Signatures Benchmarked on ASIC. ACM Trans. Embed. Comput. Syst. 16(2), 59:1–59:19 (2016)
16. McGrew, D., Curcio, M., Fluhrer, S.: Internet-Draft: Hash-Based Signatures. `https://datatracker.ietf.org/doc/draft-mcgrew-hash-sigs/` (2017)
17. NIST Computer Security Division: Post-Quantum Cryptography Standardization. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/` (2016)
18. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM J. on Computing 26(5), 1484–1509 (1997)
19. Zalka, C.: Grover's quantum searching algorithm is optimal. Phys. Rev. A60, 2746 (1999)