IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 80–114. DOI:10.46586/tches.v2023.i2.80-114

On Protecting SPHINCS+ Against Fault Attacks

Aymeric Genêt[1,2]

[1] EPFL, Lausanne, Switzerland
aymeric.genet@epfl.ch
[2] Nagra Kudelski Group, Cheseaux-sur-Lausanne, Switzerland
aymeric.genet@nagra.com

Abstract. SPHINCS$^+$ is a hash-based digital signature scheme that was selected by NIST in their post-quantum cryptography standardization process. The establishment of a universal forgery on the seminal scheme SPHINCS was shown to be feasible in practice by injecting a fault when the signing device constructs any non-top subtree. Ever since the attack has been made public, little effort was spent to protect the SPHINCS family against attacks by faults. This paper works in this direction in the context of SPHINCS$^+$ and analyzes the current algorithms that aim to prevent fault-based forgeries.

First, the paper adapts the original attack to SPHINCS$^+$ reinforced with randomized signing and extends the applicability of the attack to any combination of faulty and valid signatures. Considering the adaptation, the paper then presents a thorough analysis of the attack. In particular, the analysis shows that, with high probability, the security guarantees of SPHINCS$^+$ significantly drop when a single random bit flip occurs anywhere in the signing procedure and that the resulting faulty signature cannot be detected with the verification procedure. The paper shows both in theory and experimentally that the countermeasures based on caching the intermediate W-OTS$^+$s offer a marginally greater protection against unintentional faults, and that such countermeasures are circumvented with a tolerable number of queries in an active attack. Based on these results, the paper recommends real-world deployments of SPHINCS$^+$ to implement redundancy checks.

Keywords: SPHINCS+ · fault attack · countermeasures · post-quantum signature · hash-based cryptography

1 Introduction

In 2016, the National Institute of Standards and Technology (NIST) started a post-quantum project that aimed to standardize one or more public-key cryptographic schemes in order to complement current cryptosystems (i.e., RSA and ECDSA) with quantum-resistant alternatives. Six years later, after three rounds of meticulous examination, NIST finally delivered a verdict and selected one key encapsulation mechanism along with three digital signature schemes to be standardized while four other schemes advanced to a fourth round of evaluation.

Among the digital signature schemes that were selected for standardization by NIST, SPHINCS$^+$—a stateless hash-based digital signature scheme—was chosen thanks to its unique security assumption [GDD+22], as the entire security of the scheme relies solely on the cryptographic properties of the hash function adopted. Such a characteristic is achieved by committing with said hash function to secret values that are revealed as part of a signature depending on the bit values of the message to be signed.

As SPHINCS$^+$ is going to be standardized, the scheme will require to operate in a real-world environment in which implementations are exposed to common abuses. In

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

particular, faulty behaviors (e.g., data corruption) are of notable interest, since software or hardware failures may accidentally cause a cryptographic scheme to reveal information that, once disclosed, allows an adversary to compromise the security guarantees of the scheme. A classic example of a fault vulnerability is presented in [BDL97], in which Boneh, DeMillo, and Lipton show that a faulty RSA signature along with its message enables the recovery of the signing key. As such faults may occur naturally or be deliberately injected (see [BBB+12] for techniques), studying cryptosystems in presence of errors is therefore vital to their real-world deployment.

In 2018, Castelnovi, Martinelli, and Prest have shown in [CMP18] that SPHINCS—the seminal scheme that led to SPHINCS$^+$—was subject to a critical fault attack which was experimentally verified by Genêt et al. in [GKPM18]. The attack enables the forgery of a valid signature for *any* chosen message once both a valid and faulty signatures for the same message have been collected. This is because the scheme involves hash trees which are normally invariant from one execution to another and which are thus signed with one-time signatures. However, the introduction of a fault during their recomputation violates the invariability requirement that is necessary for the security of one-time signatures, as the fault forces the one-time signature to be used a second time, which goes against its intended purpose. This enables the existential forgery of a counterfeit tree that is validly signed by the compromised one-time signature and can then be used to forge a valid signature for any desired message. Since the attack exploits the design of the scheme rather than a flaw in its implementation, all existing implementations of SPHINCS$^+$ are impacted, as well as all other variations of SPHINCS. For example, Amiet et al. mounted the same attack on a custom hardware implementation of SPHINCS$^+$ in [ALCZ20].

Even though the attack critically impacts the security of the scheme, an effective countermeasure has not been discovered yet. In the work of [CMP18] by Castelnovi, Martinelli, and Prest, the authors failed to find a specific countermeasure and recommend classical redundancy instead. The work by Mozaffari Kermani, Azarderakhsh, and Aghaie in [KAA17] proposes specific error-detection mechanisms in hash function implementations which therefore do not entirely cover the SPHINCS$^+$ signing procedure, as well as a generic countermeasure based on recomputing hash trees with swapped nodes (i.e., also redundancy). In [GKPM18], Genêt et al. show that caching the one-time signatures of the hash trees in stateful hash-based signature schemes effectively protects against similar fault-based forgeries. This countermeasure prevents the recomputation of one-time signatures by storing the signatures of the hash trees that can still provide new signatures. However, the authors assert that the same countermeasure applied to stateless schemes is ineffective but do not provide any evidence for the claim. As a result, the extent to which SPHINCS$^+$ can be protected against fault attacks with this technique is unclear.

1.1 Contributions

The official specifications of SPHINCS$^+$ [HBD+20] present two mechanisms that address fault attacks: randomizing the signing procedure, and including the public key in the signing procedure so the resulting signatures can be verified. The current paper therefore analyzes fault injections against SPHINCS$^+$ in presence of these two mechanisms. Specifically, the contributions are the following:

- First, the paper starts by expanding the universal forgery with fault injections of Castelnovi, Martinelli, and Prest from [CMP18] to SPHINCS$^+$ when any types of faulty signatures are obtained. While the core of the attack is identical, the paper particularly shows that the attack is still applicable even if the adversary has collected two non-verifiable faulty signatures of the same W-OTS$^+$ keypair.
- Considering the extension, the paper presents a deep analysis of the universal forgery with a particular attention to the faulty signature collection. The analysis shows

that for all parameter sets the number of queries required to circumvent the first mechanism is on average within the limit of 2^{64} signatures established by NIST, and that the probability is very high that a random faulty signature is still verifiable, defeating thus the second mechanism.

- The paper then revisits the countermeasures based on caching the W-OTS$^+$ signatures in between the intermediate subtrees as suggested in previous work (see [AE17, GKPM18]) and shows that such countermeasures are ineffective, as an active adversary can always work around the caching system with a tolerable query complexity, and as a random fault still leads to an exploitable faulty signature with a marginally lower probability than without the countermeasure. This analysis is then experimentally verified on the SPHINCS$^+$ reference implementation using the ChipWhisperer framework.

As a consequence of the above points, the paper concludes that SPHINCS$^+$ is extremely sensitive to any kinds of faults, that no other current solution apart from redundancy effectively protects SPHINCS$^+$ against fault attacks, and so that all real-world deployments of SPHINCS$^+$ are recommended to implement redundancy checks to mitigate the risk. Lastly, all source code used to derive each result in the paper is made available at `https://github.com/AymericGenet/SPHINCSplus-FA`. The repository notably features a SPHINCS$^+$ implementation entirely developed in Python, as well as tools to mount the fault attack in practice.

1.2 Structure

The current paper is structured as follows: Section 2 gives an overview in high level of the principles of SPHINCS$^+$. Section 3 describes the fault attack on SPHINCS$^+$ which is analyzed in Section 4. Countermeasures are discussed and analyzed in Section 5. Finally, the paper reports experimental results of the countermeasures analyses in Section 6, and concludes with a discussion in Section 7.

2 Background

Hash-based digital signatures are cryptographic primitives that provide authentication, integrity, and non-repudiation with the sole use of cryptographic hash functions. Three categories of hash-based digital signatures are usually identified:

- One-Time Signatures (OTS) where any signing key should be used at most *once*,
- Multiple-Time Signatures (MTS) which combine multiple instances of OTSs to provide a limited number of signatures *only*,
- Few-Time Signatures (FTS) in which the security of one key pair slowly declines with the number of uses.

Because classical hash-based digital signatures require to keep track of the number of signatures used, hash-based digital signatures were considered stateful for a long time. This limitation was overcome by the scheme SPHINCS [BHH$^+$15], the first *stateless* hash-based digital signature scheme, which achieves practicality along with strong security levels by combining a large tree of MTS on top of a wide layer of FTS. Recently, SPHINCS$^+$ [BHK$^+$19] has been proposed to NIST's post-quantum standardization process as an improved version of SPHINCS.

This section presents a comprehensive summary of SPHINCS$^+$ and of all its components. For a thorough description of the SPHINCS$^+$ signature scheme, the reader is advised to read the full submission to the NIST standardization process in [HBD$^+$20].

2.1 Definitions and notations

In the original submission of SPHINCS$^+$, different[1] hash functions are used depending on the operation. Table 1 summarizes all the hash functions and pseudorandom functions involved in SPHINCS$^+$. Given a SPHINCS$^+$ signing key $\mathtt{sk}_1$, $\mathtt{sk}_2$, and public key $\mathtt{pk}_1$, $\mathtt{pk}_2$, the parameters column includes public and secret seeds (resp., $\mathtt{pk}_1$, $\mathtt{pk}_2$, and $\mathtt{sk}_2$) that make hash function calls unique per key pair, and contextual information (i.e., $\mathtt{ADRS}$, R, opt) that makes hash function calls unique per use. These are sometimes considered implicitly given in the notations. $\mathbb{B}$ denotes the set of bytes (i.e., $\mathbb{B} \cong \{0, 1, \ldots, 255\}$).

Table 1: Hash functions involved in SPHINCS$^+$.

Function	Parameters	Input	Output
$\mathbf{T}_l$	$(\mathtt{pk}_2, \mathtt{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^\alpha$	$(x_1, \ldots, x_l) \in \mathbb{B}^{ln}$	$y \in \mathbb{B}^n$
$\mathbf{F}$	$(\mathtt{pk}_2, \mathtt{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^\alpha$	$x \in \mathbb{B}^n$	$y \in \mathbb{B}^n$
$\mathbf{H}$	$(\mathtt{pk}_2, \mathtt{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^\alpha$	$(x_{\mathrm{L}}, x_{\mathrm{R}}) \in \mathbb{B}^{2n}$	$y \in \mathbb{B}^n$
$\mathbf{PRF}$	$(\mathtt{pk}_2, \mathtt{ADRS}) \in \mathbb{B}^n \times \mathbb{B}^\alpha$	$\mathtt{sk}_1 \in \mathbb{B}^n$	$s \in \mathbb{B}^n$
$\mathbf{PRF_{msg}}$	$(\mathtt{sk}_2, \mathrm{opt}) \in \mathbb{B}^{2n}$	$\mathtt{msg} \in \mathbb{B}^*$	$R \in \mathbb{B}^n$
$\mathbf{H_{msg}}$	$(\mathtt{pk}_1, R) \in \mathbb{B}^{2n}$	$\mathtt{msg} \in \mathbb{B}^*$	$(\mathtt{md}, \mathtt{ADRS}) \in \mathbb{B}^m$

In SPHINCS$^+$, a (binary hash) *tree* refers to a structure of (hash) nodes in which an initial number of 2^x *leaf values* ($x > 0$) are compressed two by two with $\mathbf{H}$ until a single value—referred to as the (tree) *root*—is reached. The process of hashing nodes two by two until reaching a single node is referred to as *treehash* (sometimes called as a subroutine, by abuse of notation). An *authentication path* refers to the nodes that are adjacent to the ones in the path from a leaf to the root and which allow a recomputation of the root (see Figure 1 for an illustration). Finally, an *address* is represented by a unique bytestring (of size $\alpha = 32$) which is composed of different fields, including notably the *tree index* that is used to uniquely address every tree involved in the scheme, and the *leaf index* to uniquely address all the leaves. When such fields are updated, the resulting addresses are differentiated using subscripts and superscripts (the corresponding field is understood in context).

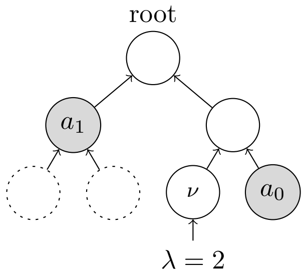


Figure 1: Example of an authentication path in a tree of four leaves. Starting from ν, the leaf indexed at $\lambda = 2$, the root can be recomputed using only auth $= \{a_0, a_1\}$.

2.2 FORS

FORS (Forest Of Random Subsets) is a few-time signature scheme which, in SPHINCS$^+$, is used to produce the actual signature of a message digest.

[1] Even though such functions are listed separately, all the hash functions involved in SPHINCS$^+$ are instantiable from a single cryptographic hash function such as SHA2-256.

A FORS instance requires the following parameters:

- n : number of bytes of security.
- η : number of bits in a message digest.
- k : number of trees.
- $t = 2^a$: number of leaves in a tree (of height a).

Note that $\eta = ka$.

Key generation. Given a SPHINCS$^+$ signing key $\mathtt{sk}_1$ and an address $\mathtt{ADRS}$, the key pair $(\textsc{sk}_i^F, \textsc{pk}_i^F)$ of the i^{th} FORS tree is computed as follows ($1 \le i \le k$):

$$\begin{array}{rcl} \textsc{sk}_i^F & \leftarrow & (s_1^{(i)}, \ldots, s_t^{(i)}), \\ \textsc{pk}_i^F & \leftarrow & \mathtt{treehash}(\mathbf{F}(s_1^{(i)}), \ldots, \mathbf{F}(s_t^{(i)})), \end{array}$$

where $s_j^{(i)} = \mathbf{PRF}(\mathtt{pk}_2, \mathtt{ADRS}_j^{(i)})(\mathtt{sk}_1)$. The overall key pair $(\textsc{sk}^F, \textsc{pk}^F)$ of a FORS is a collection of the keys of k trees:

$$\begin{array}{rcl} \textsc{sk}^F & \leftarrow & (\textsc{sk}_1^F, \ldots, \textsc{sk}_k^F), \\ \textsc{pk}^F & \leftarrow & \mathbf{T}_k(\mathtt{pk}_2, \mathtt{ADRS})(\textsc{pk}_1^F, \ldots, \textsc{pk}_k^F). \end{array}$$

Signing procedure. Given a FORS signing key $\textsc{sk}^F$, the scheme signs an η-bit digest $\mathtt{md}$ with the following procedure:

1. Split $\mathtt{md}$ into chunks $(m_1, \ldots, m_k)$ of a bits.
2. For each chunk $i \in \{1, \ldots, k\}$, compute auth$^{(i)}$ as the authentication path starting from $\mathbf{F}(s_{m_i}^{(i)})$, the leaf indexed at m_i in the i^{th} FORS subtree.
3. Return $\sigma^F = ((s_{m_1}^{(1)}, \text{auth}^{(1)}), \ldots, (s_{m_k}^{(k)}, \text{auth}^{(k)}))$.

Public key extraction. Given $\sigma^F = ((s^{(1)}, \text{auth}^{(1)}), \ldots, (s^{(k)}, \text{auth}^{(k)}))$ bound to a known message $\mathtt{md}$, the public key of the corresponding FORS can be extracted with the following procedure:

1. Split $\mathtt{md}$ into chunks $(m_1, \ldots, m_k)$ of a bits.
2. For each chunk $i \in \{1, \ldots, k\}$, recompute the public keys $\textsc{pk}_i^F$ of the i^{th} FORS tree from $\mathbf{F}(s^{(i)})$ and auth$^{(i)}$.
3. Return $\textsc{pk}^F = \mathbf{T}_k(\mathtt{pk}_2, \mathtt{ADRS})(\textsc{pk}_1^F, \ldots, \textsc{pk}_k^F)$.

The public key extraction requires a total of $k(a+1)+1$ hash calls.

2.3 W-OTS+

W-OTS$^+$ (Winternitz One-Time Signature) is a one-time signature scheme which, in SPHINCS$^+$, is used to authenticate subtrees by signing their roots (each time with a unique key pair).

W-OTS$^+$ uses a *chaining pseudorandom function* $\mathbf{C}_i(\mathtt{pk}_2, \mathtt{ADRS})(x)$ which consists of consecutively applying $\mathbf{F}$ a specified number of times $i \ge 0$ on an initial input, as illustrated in Figure 2. The *position* of an element y in the chaining function is referred to as the number i such that $\mathbf{C}_i(\mathtt{pk}_2, \mathtt{ADRS})(x) = y$. Such a position is updated in $\mathtt{ADRS}$ at each hashing step (denoted by $\mathtt{ADRS}^{(i)}$ where $0 \le i < W - 1$).

A W-OTS$^+$ instance is parameterized with:

- n : number of bytes of security.
- ω : a (short) window of bits signed at a time.
- $W = 2^\omega$: the length of the hash chains.

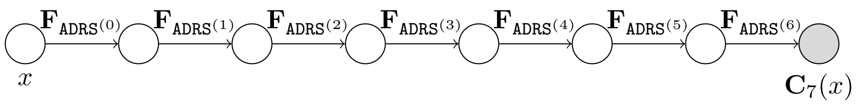

$\mathbf{F}_{\mathtt{ADRS}^{(0)}}$
$\mathbf{F}_{\mathtt{ADRS}^{(1)}}$
$\mathbf{F}_{\mathtt{ADRS}^{(2)}}$
$\mathbf{F}_{\mathtt{ADRS}^{(3)}}$
$\mathbf{F}_{\mathtt{ADRS}^{(4)}}$
$\mathbf{F}_{\mathtt{ADRS}^{(5)}}$
$\mathbf{F}_{\mathtt{ADRS}^{(6)}}$
x
$\mathbf{C}_7(x)$

Figure 2: The chaining pseudorandom function.

Moreover, the total number of n-byte elements in a W-OTS$^+$ signature is given by $\ell = \ell_1 + \ell_2$ where

$$\ell_1 = \left\lceil \frac{8n}{\omega} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log_2((W-1)\ell_1)}{\omega} \right\rfloor + 1.$$

Key generation. Given a SPHINCS$^+$ signing key $\mathtt{sk}_1$ and an address $\mathtt{ADRS}$, the generation of a W-OTS$^+$ key pair $(\textsc{sk}^W, \textsc{pk}^W)$ works as follows:

$$\begin{aligned} \textsc{sk}^W &\leftarrow (s_1, \ldots, s_\ell), & s_i &= \mathbf{PRF}(\mathtt{pk}_2, \mathtt{ADRS}_i^{(0)})(\mathtt{sk}_1), \\ \textsc{pk}^W &\leftarrow (p_1, \ldots, p_\ell), & p_i &= \mathbf{C}_{W-1}(\mathtt{pk}_2, \mathtt{ADRS}_i^{(0)})(s_i). \end{aligned}$$

Signing procedure. Given a W-OTS$^+$ signing key $\textsc{sk}^W$, the scheme signs an n-byte digest $\mathtt{msg}$ with the following procedure:

1. Split $\mathtt{msg}$ into chunks $(b_1, \ldots, b_{\ell_1})$ of ω bits.
2. Compute and split $\mathtt{csum} = \sum_{i=1}^{\ell_1} W - 1 - b_i$ into ω-bit chunks $(b_{\ell_1+1}, \ldots, b_{\ell_1+\ell_2})$.
3. Let $\sigma_i = \mathbf{C}_{b_i}(\mathtt{pk}_2, \mathtt{ADRS}_i^{(0)})(s_i)$ for $1 \le i \le \ell = \ell_1 + \ell_2$.
4. Return $\sigma^W = (\sigma_1, \ldots, \sigma_\ell)$.

Public key extraction. Given $\sigma^W = (\sigma_1, \ldots, \sigma_\ell)$ bound to a known message $\mathtt{msg}$, the public key of the corresponding W-OTS$^+$ can be extracted with the following procedure:

1. Split $\mathtt{msg}$ into chunks $(b_1, \ldots, b_{\ell_1})$ of ω bits.
2. Compute and split $\mathtt{csum} = \sum_{i=1}^{\ell_1} W - 1 - b_i$ into ω-bit chunks $(b_{\ell_1+1}, \ldots, b_{\ell_1+\ell_2})$.
3. Let $p_i = \mathbf{C}_{W-1-b_i}(\mathtt{pk}_2, \mathtt{ADRS}_i^{(b_i)})(\sigma_i)$ for $1 \le i \le \ell = \ell_1 + \ell_2$.
4. Return $\textsc{pk}^W = (p_1, \ldots, p_\ell)$.

Supposing that all b_i are uniformly distributed for $1 \le i \le \ell$, the probability that recomputing p_i requires $0 \le x < W$ hash function calls is $1/W$. As a result, the public key extraction requires $\ell(W-1)/2$ hash function calls on average.

2.4 XMSS

An XMSS (eXtended Merkle Signature Scheme), parameterized with a height h', is a multiple-time signature scheme which combines $2^{h'}$ W-OTS$^+$ in a hash tree. SPHINCS$^+$ uses a hypertree of XMSS to authenticate FORSs.

Key generation. Given a SPHINCS$^+$ signing key $\mathtt{sk}_1$ and an address $\mathtt{ADRS}$, the key generation starts by generating $2^{h'}$ W-OTS$^+$ key pairs $(\textsc{sk}_i^W, \textsc{pk}_i^W)$ (for $1 \le i \le 2^{h'}$). Thus, an overall XMSS key pair consists of:

$$\begin{aligned} \textsc{sk}^X &\leftarrow (\textsc{sk}_1^W, \ldots, \textsc{sk}_{2^{h'}}^W), \\ \textsc{pk}^X &\leftarrow \mathtt{treehash}(\mathbf{T}_\ell(\mathtt{pk}_2, \mathtt{ADRS}_1)(\textsc{pk}_1^W), \ldots, \mathbf{T}_\ell(\mathtt{pk}_2, \mathtt{ADRS}_{2^{h'}})(\textsc{pk}_{2^{h'}}^W)). \end{aligned}$$

Signing procedure. Given an XMSS signing key $\textsc{sk}^X$, the scheme signs an n-byte digest $\texttt{msg}$ at leaf index $1 \leq \lambda \leq 2^{h'}$ with the following procedure:

1. Use $\textsc{sk}^W_\lambda$ to produce σ^W_λ; the W-OTS$^+$ signature of $\texttt{msg}$.
2. Compute the authentication path auth_λ from the leaf index λ.
3. Return $\sigma^X = (\sigma^W_\lambda, \text{auth}_\lambda)$.

Note that each leaf index can be used to sign at most one message.

Public key extraction. Given $\sigma^X = (\sigma^W_\lambda, \text{auth}_\lambda)$ bound to a known digest $\texttt{msg}$ at leaf index $1 \leq \lambda \leq 2^{h'}$, the public key of the corresponding XMSS can be extracted with the following procedure:

1. Extract the W-OTS$^+$ public key $\textsc{pk}^W_\lambda$ from σ^W_λ using the message $\texttt{msg}$.
2. Recompute the XMSS public key $\textsc{pk}^X$ from $\mathbf{T}_\ell(\texttt{pk}_2, \texttt{ADRS}_\lambda)(\textsc{pk}^W_\lambda)$ and auth_λ.
3. Return $\textsc{pk}^X$.

The public key extraction requires $\ell(W-1)/2 + 1 + h'$ hash function calls on average, since the procedure depends on the extraction of a W-OTS$^+$ public key.

2.5 Hypertree

In the context of SPHINCS$^+$, a hypertree consists of a tree of XMSS key pairs in which the XMSSs above sign the XMSSs below (with respect to a tree with the root at the top).

A hypertree is parameterized with a height h', where

- h : total height of the hypertree,
- d : number of layers in the hypertree.

Key generation. Given a SPHINCS$^+$ signing key $\texttt{sk}_1$, the public key of the hypertree consists of the public key of the top-most XMSS $\textsc{pk}^X_{d-1}$ at layer $d-1$ (addressed at $\tau_{d-1} = 0$) generated with $\texttt{sk}_1$:

$$\begin{array}{rcl} \textsc{sk}^{HT} & \leftarrow & \texttt{sk}_1, \\ \textsc{pk}^{HT} & \leftarrow & \textsc{pk}^X_{d-1}. \end{array}$$

Addresses derivation. Due to their structure, the addresses of the subtrees above can be entirely derived from the address of the subtree below. Let τ_i be the tree index of an XMSS at layer $0 \leq i < d-1$, such an XMSS is signed with the XMSS at tree index τ_{i+1} and leaf index λ_{i+1} derived as follows:

$$\begin{cases} \tau_{i+1} & = & \text{the } h - h'(i+1) \text{ most significant bits of } \tau_i, \\ \lambda_{i+1} & = & \text{the } h' \text{ least significant bits of } \tau_i. \end{cases}$$

All addresses involved in the above XMSSs are reconstructed from these indices.

Signing procedure. Given a hypertree signing key $\textsc{sk}^{HT}$, the scheme signs an n-byte digest r at hyperleaf index $1 \leq \lambda \leq 2^h$ with the following procedure:

1. For $0 \leq i < d$:
   (a) Derive τ_i, λ_i from τ_{i-1} (starting with $\tau_{-1} = \lambda$).
   (b) Generate the XMSS key pair $(\textsc{sk}^X_i, \textsc{pk}^X_i)$ at the address corresponding to τ_i.
   (c) Sign r with $\textsc{sk}^X_i$ using λ_i as leaf index to produce σ_i and update r with $\textsc{pk}^X_i$.
2. Return $\sigma^{HT} = (\sigma_0, \ldots, \sigma_{d-1})$.

Note that each hyperleaf index should be used to sign at most one message.

Public key extraction. Given a hypertree signature $\sigma^{HT} = (\sigma_0^X, \ldots, \sigma_{d-1}^X)$ which corresponds to a known digest r at hyperleaf index $1 \leq \lambda \leq 2^h$, the public key can be extracted with the following procedure:

1. For $0 \leq i < d$:
   (a) Derive τ_i, λ_i from τ_{i-1} (starting with $\tau_{-1} = \lambda$).
   (b) Extract the XMSS public key $\textsc{pk}_i^X$ from σ_i^X using `msg` at the address corresponding to τ_i and using the λ_i as leaf index.
   (c) Update r with $\textsc{pk}_i^X$.
2. Return the last r computed, i.e, $\textsc{pk}_{d-1}^X$.

The public key extraction requires $d(\ell(W-1)/2 + 1 + h')$ hash function calls on average.

2.6 SPHINCS+

SPHINCS$^+$ is a stateless signature scheme which combines FORSs with a hypertree, as illustrated in Figure 3.

SPHINCS$^+$ is parameterized with the following:

• n	: number of bytes of security.
• m	: number of bytes of the digest.
• k, $t = 2^a$	: FORS parameters.
• $W = 2^\omega$	: W-OTS$^+$ parameters.
• $h' = h/d$	: hypertree parameters.

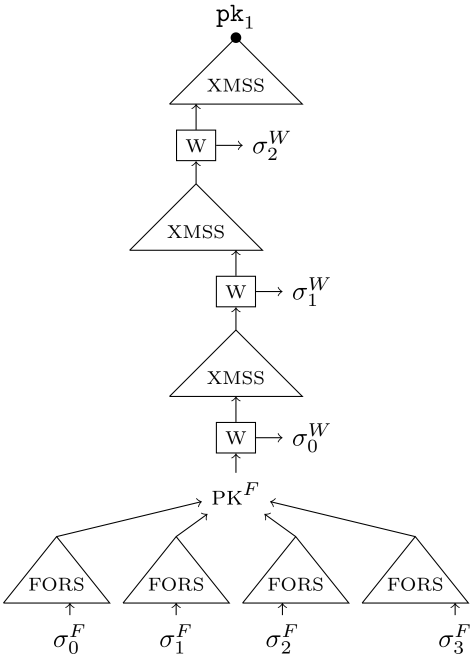


Figure 3: The SPHINCS$^+$ structure.

Key generation. The SPHINCS$^+$ signing key consists of two secret n-byte seeds $\texttt{sk}_1$ and $\texttt{sk}_2$ picked uniformly at random:

- $\texttt{sk}_1$ is used to derive the key pairs of all the hash-based instances involved in the scheme.
- $\texttt{sk}_2$ is used to choose a starting FORS in an unpredictable way.

The SPHINCS$^+$ public key $\texttt{pk}_1$ consists of the public key of hypertree as well as a public seed $\texttt{pk}_2$ that makes hash function calls unique per user.

Signing procedure. Given a SPHINCS$^+$ signing key $(\texttt{sk}_1, \texttt{sk}_2)$, the scheme signs a message $\texttt{msg}$ of arbitrary bit length with the following procedure:

1. Generate $R = \mathbf{PRF_{msg}}(\texttt{sk}_2, \text{opt})(\texttt{msg})$ where opt $\in \mathbb{B}^n$ is uniformly drawn at random to enable *randomized signing.*
2. Compute $(\texttt{md}, \texttt{ADRS}) = \mathbf{H_{msg}}(\texttt{pk}_1, R)(\texttt{msg})$.
3. Sign $\texttt{md}$ with the FORS at $\texttt{ADRS}$ using $\texttt{sk}_1$ to produce σ^F, and let $\textsc{pk}^F$ be the FORS public key.
4. Sign $\textsc{pk}^F$ with the hypertree to produce $\sigma^{HT} = (\sigma_0^X, \ldots, \sigma_{d-1}^X)$.
5. Return $\Sigma = (R, \sigma^F, \sigma^{HT})$.

Verification procedure. Given a SPHINCS$^+$ public key $\texttt{pk}_1$, the scheme verifies that a SPHINCS$^+$ signature $\Sigma = (R, \sigma^F, \sigma^{HT})$ corresponds to the message $\texttt{msg}$ with the following procedure:

1. Compute $(\texttt{md}, \texttt{ADRS}) = \mathbf{H_{msg}}(\texttt{pk}_1, R)(\texttt{msg})$.
2. Extract $\textsc{pk}^F$, the public key of the FORS at $\texttt{ADRS}$, from $\texttt{md}$ and σ^F.
3. Extract $\textsc{pk}^{HT}$, the public key of the hypertree at the leaf index given by $\texttt{ADRS}$, from $\textsc{pk}^F$ and $\sigma^{HT} = (\sigma_0^X, \ldots, \sigma_{d-1}^X)$.
4. Return true if $\textsc{pk}^{HT} = \texttt{pk}_1$, false otherwise.

Summing up all the calls from all the components, the verification procedure requires $1 + (k(a+1)+1) + d(\ell(W-1)/2 + 1 + h')$ hash function calls on average.

3 Fault attack

In their original attack in [CMP18], Castelnovi, Martinelli, and Prest present a fault attack that forces a W-OTS$^+$ key pair to sign a corrupted message by injecting a fault during the construction of any non-top subtree. Along with the valid (i.e., non-faulted) signature of the subtree, the resulting W-OTS$^+$ faulty signature is used to compromise the corresponding W-OTS$^+$ key pair under a two-message attack and provide a valid signature for another subtree for which the secrets are known. This process—similar to a *tree grafting*—enables the forgery of an overall signature for any message.

This section expands the fault attack from Castelnovi, Martinelli, and Prest to any combination of valid and faulty signatures obtained.

Attack preliminaries

Target. In the following, we consider a target device which runs any instance of SPHINCS$^+$ with a fixed and unknown signing key, but a known public key. Furthermore, such instance is supposed hardened with randomized signing (as described in Section 2.6) using a source of true randomness.

Adversarial model. The threat model considers an adversary who has access to a number of valid and faulty signatures (along with their messages) produced by the target device. The goal of the adversary is to forge a SPHINCS$^+$ signature that verifies any chosen message under the target device's public key.

Fault characteristics. The faulty signatures consist of outputs from the target device when a single unconstrained corruption of one-to-many bits occurs in any value involved in the entire SPHINCS$^+$ signing procedure. Such an outcome can happen due to the accidental or intentional effect of, e.g., the target device overheating [BBB+12], voltage disturbances [BBB+12], or row-hammer [KDK+14]. The typical use cases where the fault model is relevant include all scenarios in which a large number of signatures may be queried, such as with embedded devices, or TLS.

Due to their significant cost compared to other instructions, the fault is further assumed to occur in a hash function call. Moreover, such a fault is supposed to cause the output of the hash function to completely deviate from its intended value and be uniformly drawn at random in the co-domain of the hash function. This is aligned with the avalanche property of cryptographic hash functions in which a single bit flip early in the procedure causes an extremely different output. Besides, even if a bit flip occurs in the output of a hash function, such a bit flip will propagate in subsequent hash function calls and eventually cause uniform outputs (unless, of course, the fault hits the output of the very last hash function call of the hash structure).

3.1 Signatures collection

In a first phase, the adversary requires to collect both valid and faulty SPHINCS$^+$ signatures from the target device.

Verifiability. Distinguishing between valid and faulty signatures is not straightforward, as faulty signatures can still verify their message under the right public key. Instead, we differentiate two types of signatures:

1. *Verifiable signatures*: signatures that still verify their associated message under the public key of the device.

   These signatures generally correspond to valid signatures, but can also correspond to faulty signatures for which a fault occurred during the derivation of any node in an authentication path. This property enables the correct rederivation of all the subtree roots that were involved in the signature, as well as a necessarily valid top part.

2. *Non-verifiable signatures*: signatures that do not verify their associated message under the public key of the device.

   While these signatures are necessarily faulty, there are two further distinctions of non-verifiable signatures that can be made:

   - *Non-verifiable but correct*: all W-OTS$^+$ signatures still correspond to actual W-OTS$^+$ values at correct addresses.

     This type of signatures is obtained when a fault occurs on the path from the leaf to the root of a subtree. No subtree root can be recovered for sure from this kind of signature (unless the layer index at which the fault occurred is known).

   - *Non-verifiable and incorrect*: the W-OTS$^+$ signatures do not correspond to W-OTS$^+$ values.

     This type of signatures is typically obtained when the entire output is corrupted. These signatures do not divulge any information and need to be discarded.

Fault exploitability. In addition to the above nomenclature, a faulty signature is said to be *exploitable* when the resulting signature contains a faulty W-OTS$^+$ signature which discloses unintentional secret values of the associated W-OTS$^+$ key pair. Such an outcome occurs only when a fault hits any non-top subtree (including the ones in FORS).

An exploitable signature alone is not sufficient to compromise a W-OTS$^+$ key pair. At least one more signature of the same W-OTS$^+$ (such as the valid one) is needed as well. As a result, the next step of the attack aims to determine the compromised W-OTS$^+$s by identifying the different signatures that correspond to a same key pair.

Compromised W-OTS$^+$ identification. Once valid and faulty SPHINCS$^+$ signatures $\{\Sigma_i : 0 \leq i < N\}$ have been collected, the W-OTS$^+$ signatures in the SPHINCS$^+$ signatures need to be arranged by layer and address:

1. Derive the `ADRS` of each W-OTS$^+$ signature in all Σ_i $(0 \leq i < N)$ from the hypertree leaf index obtained in $(\_, \texttt{ADRS}) = \mathbf{H_{msg}}(R)(\texttt{msg})$ (see Section 2.5).
2. Map all the W-OTS$^+$ signatures in Σ_i to their respective layer and `ADRS`.

If two or more different W-OTS$^+$ signatures are mapped to a same `ADRS` at the end of the arrangement, then the corresponding W-OTS$^+$ key pair is said to be *compromised*. In this case, such collection of W-OTS$^+$ signatures is referred to as the *faulty W-OTS$^+$ signatures* and are denoted by $(\hat{\sigma}^{(i)})_{i=0}^{M}$, while their respective full SPHINCS$^+$ signatures are denoted by $(\hat{\Sigma}^{(i)} : \hat{\sigma}^{(i)} \in \hat{\Sigma}^{(i)})_{i=0}^{M}$. We denote their layer index[2] by $l^* \in \{0, \ldots, d\}$, and denote their address by $\texttt{ADRS}^*$. Finally, we refer to all layers below (resp. above) the faulted layer as the *bottom part* (resp. as the *top part*) of the hypertree, as illustrated in Figure 4.

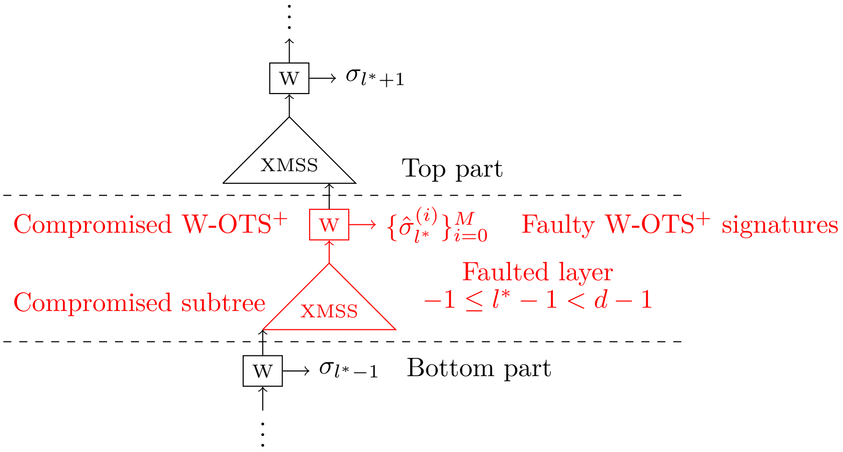


Figure 4: Terminology used throughout the description of the attack.

3.2 Faulty signatures processing

The next step processes the faulty SPHINCS$^+$ signatures identified in the previous section to extract the information that enables the universal forgery.

Secret values identification. As the elements in a W-OTS$^+$ signature correspond to secret values associated to chunks of ω bits (see Section 2.3), the following process aims to identify the value of the chunks that are associated to each element.

[2] Note that $l^* = 0$ means that the fault has hit the FORS layer, and that $l^* = d$ means that the fault has hit the top XMSS which does not lead to an exploitable signature.

Such a process depends on the types of signatures obtained:

- **Case 1**: At least one verifiable signature is available.

  Given a verifiable signature, the correct public key of the compromised W-OTS$^+$ can be extracted from the SPHINCS$^+$ signature (see Section 2.3). Note that the integrity of the extracted public key must be preserved even when its corresponding subtree was faulted, as the signature verifies the extracted key under the correct SPHINCS$^+$ public key.

  The extracted W-OTS$^+$ public key can then be used to identify all the secret values in the other signatures, including the non-verifiable (but valid) ones. Strictly speaking, given the W-OTS$^+$ public key $\text{PK}^W = (p_1, \ldots, p_\ell)$ and any type of W-OTS$^+$ signature $\hat{\sigma}^W = (\hat{\sigma}_1, \ldots, \hat{\sigma}_\ell)$, the secret values are identified with the following exhaustive search:

  1. Create the next ω-bit chunk b_i corresponding to $\hat{\sigma}_i$ $(1 \le i \le \ell)$.
  2. Check that the value is correct with $\mathbf{C}_{W-1-b_i}(\texttt{pk}_2, \texttt{ADRS}^{*(b_i)})(\hat{\sigma}_i) = p_i$.

  If no value leads to the W-OTS$^+$ public key element, then the $\hat{\sigma}^W$ is *incorrect.*

  Complexity. Extracting the public key of the compromised W-OTS$^+$ is equivalent to running a truncated SPHINCS$^+$ verification procedure with l^* layers (see Section 2.6), which therefore amounts to an average number of hash function calls of:

  $$2 + k(a+1) + l^*(\ell(W-1)/2 + 1 + h').$$

  Now, suppose that the ω-bit chunks $(\hat{b}_1^{(i)}, \ldots, \hat{b}_\ell^{(i)})$ that correspond to the W-OTS$^+$ signature $(\hat{\sigma}_1^{(i)}, \ldots, \hat{\sigma}_\ell^{(i)})$ are uniformly distributed. For $1 \le j \le \ell$, finding the value of the chunk $\hat{b}_j^{(i)}$ that corresponds to $\hat{\sigma}_j$ requires $W-1-x$ applications of $\mathbf{F}$ for a hypothesized initial position $0 \le x \le W-1$ until the resulting value equals p_i. As each value occurs with probability $1/W$, the average number of hash function calls is:

  $$\sum_{x=0}^{W-1} \left(\frac{1}{W}\right)(W-1-x) = (W-1)/2.$$

  As there are ℓ blocks in each $\hat{\sigma}^W$, the overall number of hash function calls for this case is $\ell(W-1)/2$.

- **Case 2**: Only non-verifiable signatures are available.

  Since none of the subtree roots can be recovered for sure, the adversary cannot extract the compromised W-OTS$^+$ public key from the non-verifiable signatures. However, the adversary can determine the positions of each W-OTS$^+$ value by using one value as a reference for the other.

  In other words, given a pair of different W-OTS$^+$ signatures, i.e., $(\hat{\sigma}^{(0)}, \hat{\sigma}^{(1)})$ where $\hat{\sigma}^{(0)} = (\hat{\sigma}_1^{(0)}, \ldots, \hat{\sigma}_\ell^{(0)})$ and $\hat{\sigma}^{(1)} = (\hat{\sigma}_1^{(1)}, \ldots, \hat{\sigma}_\ell^{(1)})$, consider the two values $\hat{\sigma}_i^{(0)}, \hat{\sigma}_i^{(1)}$ at a same index $1 \le i \le \ell$. There are two possibilities:

  - $\hat{\sigma}_i^{(0)} \ne \hat{\sigma}_i^{(1)}$: in this case, if both signatures are *correct*, then there must exist positions $0 \le u < v < W$ such that

    $$\mathbf{C}_v(\texttt{pk}_2, \texttt{ADRS}^{*(u)})(\hat{\sigma}_i^{(0)}) = \hat{\sigma}_i^{(1)}, \text{ or } \mathbf{C}_v(\texttt{pk}_2, \texttt{ADRS}^{*(u)})(\hat{\sigma}_i^{(1)}) = \hat{\sigma}_i^{(0)}.$$

    The above property enables confirming guesses on u and v, which directly leads to the i^{th} ω-bit chunk of both roots, since the hash applications use different addresses at each step of the chaining pseudorandom function. As a result, the values are extracted as follows:

1. Create the next ω-bit chunks u, v respectively corresponding to $\hat{\sigma}_i^{(0)}, \hat{\sigma}_i^{(1)}$ $(1 \leq i \leq \ell)$.
2. Check that the values are correct with $\mathbf{C}_v(\mathtt{pk}_2, \mathtt{ADRS}^{*(u)})(\hat{\sigma}_i^{(0)}) = \hat{\sigma}_i^{(1)}$ or $\mathbf{C}_v(\mathtt{pk}_2, \mathtt{ADRS}^{*(u)})(\hat{\sigma}_i^{(1)}) = \hat{\sigma}_i^{(0)}$.

If no such u and v exist, then at least one of the signatures is *incorrect.*

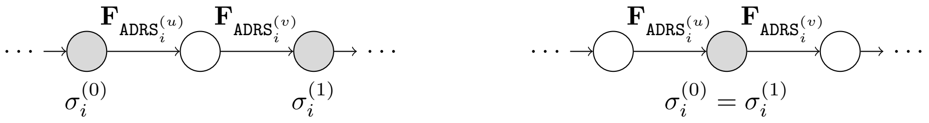
Complexity. Supposing that all chunks are uniformly distributed, the probability that $\hat{b}_j^{(0)}$ and $\hat{b}_j^{(1)}$ take different values is $1/(W(W-1))$. Since, for a fixed u, the exhaustive search on v can apply $\mathbf{F}$ on the previous hash result, the average number of hash calls is:

$$\sum_{x=1}^{W(W-1)} \left(\frac{1}{W(W-1)}\right) x = \frac{W(W-1)+1}{2}.$$

- $\hat{\sigma}_i^{(0)} = \hat{\sigma}_i^{(1)}$: in this case, if both signatures are *correct*, then the two values must correspond to the same ω-bit chunk, but of unknown position in the chaining pseudorandom function. Another signature with a different value at index i is required to identify the value of the chunks.

  If there are still chunks of unknown positions at the end of the secret values identification, such positions can be retrieved while extracting the top part of the SPHINCS$^+$ signature (see below).

An illustration for the two possibilities for $\hat{\sigma}_i^{(0)}$, $\hat{\sigma}_i^{(1)}$ in a chaining pseudorandom function is shown in Figure 5.


$\mathbf{F}_{\mathtt{ADRS}_i^{(u)}}$
$\mathbf{F}_{\mathtt{ADRS}_i^{(v)}}$
$\cdots$
$\cdots$
$\sigma_i^{(0)}$
$\sigma_i^{(1)}$
$\mathbf{F}_{\mathtt{ADRS}_i^{(u)}}$
$\mathbf{F}_{\mathtt{ADRS}_i^{(v)}}$
$\cdots$
$\cdots$
$\sigma_i^{(0)} = \sigma_i^{(1)}$

(a) Different positions in $\mathbf{C}$.

(b) Same (unknown) position in $\mathbf{C}$.

Figure 5: Identifying W-OTS$^+$ values within non-verifiable signatures.

The above process is applied to all faulty W-OTS$^+$ signatures in order to retrieve as many secret values as possible to forge a signature for a variety of different ω-bit chunks.

Note that since the values at lower positions in the chaining function enable the recomputation of values at higher positions, the identification of W-OTS$^+$ secret values can keep track of the values at the lowest positions only. As a result, in the following, we refer to the lowest positions learnt by the secret extraction as the *most secret* elements which are denoted by $(\tilde{\theta}_1, \ldots, \tilde{\theta}_\ell)$ and are respectively associated to the ω-bit chunks of $(\tilde{b}_1, \ldots, \tilde{b}_\ell)$.

Top part extraction. Extracting a valid top part is required so that the verification of the forged SPHINCS$^+$ signature leads to the target device's public key.

The extraction considers the case in which multiple top parts are available due to our fault model. Under these circumstances, all the top parts available need to be tried out starting from the compromised W-OTS$^+$ public key until one leads to the target device's public key.

Let $(\sigma X_{l^*+1}^{(i)}, \ldots, \sigma X_{d-1}^{(i)})$ be the top part of the SPHINCS$^+$ signature $\hat{\Sigma}_i$ $(0 \leq i < M)$, and $\mathtt{pk}_1$ be the SPHINCS$^+$ public key, and suppose that the compromised W-OTS$^+$ public key $\textsc{pk}^W = (p_1, \ldots, p_\ell)$ has been successfully extracted (see above):

1. Extract the hypertree public key $\textsc{pk}^{HT}$ from $\mathbf{T}_\ell(\texttt{pk}_2, \texttt{ADRS}^*)(\textsc{pk}^W)$ and the XMSS signatures $(\sigma^{X}{}^{(i)}_{l^*+1}, \ldots, \sigma^{X}{}^{(i)}_{d-1})$ (see Section 2.5).
2. Check that $\textsc{pk}^{HT} = \texttt{pk}_1$.

In case there were still ω-bit chunks of unknown values at the end of the secrets extraction, such chunks may be identified during this part by guessing all of the unknown chunks at once, deriving the corresponding W-OTS$^+$ public key, and trying this public key with the above steps. Such a process both confirms the values of the unknown chunks, and extracts a valid top part of the signature.

Complexity. Verifying that a selected top part is valid requires a truncated SPHINCS$^+$ verification procedure starting from the compromised W-OTS$^+$. Along with all the top parts available, there may be chunks that need to be exhaustively searched in case no verifiable signature was available (see above). Supposing that the blocks are uniformly distributed, the probability that, for a fixed index $1 \le j \le \ell$, all W-OTS$^+$ elements $\hat{\sigma}_j^{(i)}$ are the same is $1/W^{M-1}$. Therefore, on average, the number of chunks of unknown value is:

$$\mathbb{E}(\text{Non-id. chunks}) = \ell / W^M.$$

Given $\textsc{pk}^W = (p_1, \ldots, p_\ell)$, each trial requires one application of $\mathbf{T}_\ell$ and the recomputation of the root of the XMSS right above the faulted layer, as well as the full public key extraction of $(d-1) - l^*$ XMSS public keys. Therefore, the average number of hash function calls is:

$$1 + h' + (d - l^* - 1)(\ell(W-1)/2 + 1 + h').$$

3.3 Tree grafting

Once the most secret values of a compromised W-OTS$^+$ key pair were successfully extracted from the faulty signatures, the adversary aims to *graft* a subtree (or a forest) to the extracted top part, i.e., find another XMSS (or FORS) for which a valid W-OTS$^+$ signature can be forged in order to spoof the compromised instance at its own address.

During this step, the adversary attempts to sign the root of a forged FORS or XMSS with the W-OTS$^+$ secret values at disposal. Let $(\tilde{\theta}_1, \ldots, \tilde{\theta}_\ell)$ be the most secret W-OTS$^+$ values extracted which correspond to the ω-bit chunks $(\tilde{b}_1, \ldots, \tilde{b}_\ell)$, the grafting procedure repeats the following until successful:

1. Draw $\texttt{sk}' \in \mathbb{B}^n$ uniformly at random.
2. If $l^* = 0$, create a FORS of public key r' with $\texttt{sk}'$ at $\texttt{ADRS}^*$ (see Section 2.2), else, create an XMSS of public key r' with $\texttt{sk}'$ at $\texttt{ADRS}^*$ (see Section 2.4).
3. Split r' and its checksum into chunks $(r'_1, \ldots, r'_\ell)$ of ω bits.
4. Check that $r'_i \le b_i$ for all $1 \le i \le \ell$.

Once found, the secret key of the grafted subtree is $\texttt{sk}'$ and its signature is:

$$\sigma'^X_{l^*} = (\mathbf{C}_{r'_1 - \tilde{b}_1}(\texttt{pk}_2, \texttt{ADRS}^{*(\tilde{b}_1)})(\tilde{\theta}_1), \ldots, \mathbf{C}_{r'_\ell - \tilde{b}_\ell}(\texttt{pk}_2, \texttt{ADRS}^{*(\tilde{b}_\ell)})(\tilde{\theta}_\ell)).$$

Complexity. The tree grafting depends on the layer hit:

- In case a FORS needs to be forged ($l^* = 0$), the public key derivation amounts to k generations of FORS trees, each of them requiring a treehash procedure of height $\log_2(t) = a$, in addition to a final application of $\mathbf{T}_k$ with all the FORS tree roots:

$$k\left(t + \sum_{i=0}^{a} 2^{i-1}\right) + 1 = k(3t-1) + 1.$$

- In case an XMSS needs to be forged ($1 \leq l^* \leq d-1$), the public key derivation amounts to $2^{h'}$ W-OTS$^+$ public keys generation and a treehash procedure of height h':

$$2^{h'}(\ell + \ell(W-1)+1) + \sum_{i=1}^{h'} 2^{i-1} = 2^{h'}(\ell W + 2) - 1.$$

The probability that one attempt is successful is given by an extension of the work of Bruinderink and Hülsing in [BH17]. Given $M > 1$ different W-OTS$^+$ signatures, supposing that the chunks $(b_1, \ldots, b_\ell)$ are uniformly[3] distributed, each chunk x occurs with probability $1/W$ and enables the forgery of all chunks from x to $W-1$. Thus, the overall probability that the root of a forged XMSS can be signed is:

$$\mathbb{P}(\text{Grafting}) \leq \frac{1}{W^\ell}\left(\sum_{x=0}^{W-1}\left(1 - \left(\frac{W-1-x}{W}\right)^M\right)\right)^\ell \qquad \approx e^{-\ell/(M+1)} + \mathcal{O}(1).$$

3.4 Path seeking

The SPHINCS$^+$ signing procedure follows a path in the hypertree depending on the message and a value R. As a result, the adversary requires to find an adequate value R that makes the forged signature visit the compromised subtree.

Straightforwardly, given the message `msg'` to be maliciously signed, the value R is brute-forced until the corresponding tree index at layer l^* is the same as the grafted subtree:

1. Draw $R' \in \mathbb{B}^n$ uniformly at random.
2. Check that the hypertree leaf index in $(\_, \texttt{ADRS}) = \mathbf{H}_{\texttt{msg}}(\texttt{pk}_1, R')(\texttt{msg'})$ leads to the tree index of the grafted subtree (see Section 2.5).

While a single grafted subtree allows the adversary to forge valid SPHINCS$^+$ signatures for as many messages as desired, note that path seeking depends on the message and therefore needs to be repeated for each new message.

Complexity. Finding R' is equivalent to an exhaustive search of n bytes such that the $h - h'l^*$ most significant bits of the tree index give the index of the grafted subtree (see Section 2.5). Each trial requires only a single hash function application, and its probability of success is simply $2^{-(h-h'l^*)}$. Consequently, the adversary requires $2^{h-h'l^*}$ hash function calls on average to find an appropriate value for R'.

3.5 Universal forgery

Piecing everything together, the adversary uses the grafted subtree and the value R' to forge a bottom part of the signature, then plugs the extracted top part onto the forged part to craft a valid signature for the malicious message (selected in Section 3.4). The procedure goes as follows:

1. Generate arbitrary key pairs to forge $(\sigma'^F, \sigma_0'^X, \ldots, \sigma_{l^*-1}'^X)$, i.e., all the signatures in the layers below the grafted subtree (see Section 2.2 and Section 2.5).
2. Sign $\sigma_{l^*-1}'^X$ with the grafted XMSS at address $\texttt{ADRS}^*$ using $\texttt{sk}'$ (see Section 2.4).
3. Copy the top part for the rest of the signatures.

The final signature that verifies $\texttt{msg}'$ under the device's public key is therefore:

$$\Sigma' = (\underset{\substack{\uparrow \\ \text{sought} \\ \text{Section 3.4}}}{R'}, \underbrace{\sigma'^F, \sigma_0'^X, \ldots, \sigma_{l^*-1}'^X}_{\substack{\text{forged} \\ \text{Section 3.5}}}, \underset{\substack{\uparrow \\ \text{grafted} \\ \text{Section 3.3}}}{\sigma_{l^*}'^X}, \underbrace{\sigma_{l^*+1}^X, \ldots, \sigma_{d-1}^X}_{\substack{\text{extracted} \\ \text{Section 3.2}}}).$$

[3] We call attention to the fact that the uniform hypothesis of the blocks $(b_{\ell_1+1}, \ldots, b_{\ell_1+\ell_2})$ is not rigorous as these blocks are actually sums of uniform random variables. However, simulations in [CMP18, GKPM18] show that such a discrepancy is tolerable for our use cases.

The average computational complexity of each step in the universal forgery is shown in Table 2 for all SPHINCS$^+$ parameters sets. These numbers suggest that the fault attack is feasible in all scenarios, although the number of required hashes varies significantly depending on the specific layer targeted by the attack. However, even though the reported numbers seem high, the overall number of required hashes can still be attainable in practice[4]. This result is especially important as the fault attack can therefore be successful even if the fault is uncontrolled. The latter will be analyzed in the next section.

Table 2: Average complexity of each step of the universal forgery for all SPHINCS$^+$ parameters (the 'f' instances stand for "fast", while the 's' instances stand for "small").

	Processing (Section 3.2)						Path seeking (Section 3.4)				
	Case 1	Case 2	$\mathbb{E}$(Non-id. chunks)					(hashes)			
	(hashes)	(hashes)	$M =$	2	3	4	$l^* =$	0	1	…	$d-1$
128s	$2^{8.04}$	$2^{12.04}$		2.12	0.14	0.01		2^{64}	2^{56}	…	2^{8}
128f	$2^{8.04}$	$2^{12.04}$		2.12	0.14	0.01		2^{60}	2^{57}	…	2^{3}
192s	$2^{8.58}$	$2^{12.59}$		3.19	0.20	0.01		2^{64}	2^{56}	…	2^{8}
192f	$2^{8.58}$	$2^{12.59}$		3.19	0.20	0.01		2^{66}	2^{63}	…	2^{3}
256s	$2^{8.97}$	$2^{12.98}$		4.19	0.26	0.01		2^{64}	2^{56}	…	2^{8}
256f	$2^{8.97}$	$2^{12.98}$		4.19	0.26	0.01		2^{68}	2^{64}	…	2^{4}

	Grafting (Section 3.3)										
		FORS (hashes)					XMSS (hashes)				
	$M =$	2	4	8	16	32	2	4	8	16	32
128s		$2^{38.11}$	$2^{29.32}$	$2^{24.25}$	$2^{21.59}$	$2^{20.36}$	$2^{35.34}$	$2^{26.55}$	$2^{21.48}$	$2^{18.81}$	$2^{17.58}$
128f		$2^{33.70}$	$2^{24.90}$	$2^{19.84}$	$2^{17.17}$	$2^{15.94}$	$2^{30.34}$	$2^{21.55}$	$2^{16.48}$	$2^{13.81}$	$2^{12.58}$
192s		$2^{47.92}$	$2^{35.11}$	$2^{27.72}$	$2^{23.84}$	$2^{22.05}$	$2^{44.21}$	$2^{31.39}$	$2^{24.01}$	$2^{20.12}$	$2^{18.33}$
192f		$2^{41.16}$	$2^{28.34}$	$2^{20.96}$	$2^{17.07}$	$2^{15.28}$	$2^{39.21}$	$2^{26.39}$	$2^{19.01}$	$2^{15.12}$	$2^{13.33}$
256s		$2^{54.90}$	$2^{38.06}$	$2^{28.36}$	$2^{23.26}$	$2^{20.91}$	$2^{52.92}$	$2^{36.09}$	$2^{26.39}$	$2^{21.28}$	$2^{18.93}$
256f		$2^{51.35}$	$2^{34.51}$	$2^{24.81}$	$2^{19.71}$	$2^{17.35}$	$2^{48.92}$	$2^{32.09}$	$2^{22.39}$	$2^{17.28}$	$2^{14.93}$

4 Attack analysis

This section analyzes the fault attack described in Section 3.

4.1 Fault analysis

Since our fault model considers that faults only affect the results of hash functions, the following counts the number of hash function calls in the entire SPHINCS$^+$ signing procedure to determine the proportion of calls that, when faulted, lead to an exploitable or a verifiable faulty signature.

1. **Path derivation:** $R = \mathbf{PRF}_{\mathbf{msg}}(\mathtt{sk}_2, \mathrm{opt})(\mathtt{msg})$.
   - *Total hash function calls:* 1.
   - *Fault exploitability:* No.
   - *Signature verifiability:* The resulting signature is *verifiable* (even valid), as R is anyway included in the signature and the result of a random selection.

[4] For reference, for SHA2-256, an Nvidia RTX 3090 is reported to perform $2^{36.95}$ hashes per second (see [Onl22]). The actual performance may vary since, in our use cases, the results of previous hash function calls need to be used as inputs to the next ones.

2. **Digest and initial address:** $(\mathtt{md}, \mathtt{ADRS}) = \mathbf{H_{msg}}(\mathtt{pk}_1, R)(\mathtt{msg})$.
   - *Total hash function calls:* 1.
   - *Fault exploitability:* No.
   - *Signature verifiability:* The resulting signature is *non-verifiable and incorrect*, as an improper FORS is used to sign an improper digest.

3. **FORS signature** (i.e., $l^* = 0$).
   - *Total hash function calls:* $\#\mathrm{Total}^F = k(3t-1)+1$.
   - *Fault exploitability:* Yes.
   - *Signature verifiability:* The verifiability of the resulting signature depends on the location of the fault in the subtrees:
     - A *verifiable signature* is obtained when a fault hits any value involved in an authentication path of a FORS tree. Each authentication path requires the derivation of $t-1$ secret values, as well as $2^{a-i}-1$ nodes in level $0 \leq i \leq a$ of a tree. As there are k trees, this amounts to a total number of verifiable signatures of:

       $$\#\mathrm{Verif}^F = k\left((t-1) + \sum_{i=0}^{a}(2^{a-i}-1)\right) = k(3t-a-3).$$

     - A *non-verifiable but correct* signature is obtained when a fault hits any value on the path from a leaf to the root of a FORS tree. The values in a path consist of the secret leaf derivation, in addition to a single node in all levels of a tree, and the computation of the FORS public key. As there are k trees of $t = 2^a$ leaves, this amounts to a total number of non-verifiable but correct signatures of:

       $$\#\mathrm{Non\text{-}verif}^F = k\left(1 + \sum_{i=0}^{a} 1\right) + 1 = k(a+2)+1.$$

4. **XMSS signature** at a non-top layer (i.e., $1 \leq l^* < d$).
   - *Total hash function calls:* $\#\mathrm{Total}^X = 2^{h'}(\ell W + 2) - 1$.
   - *Fault exploitability:* Yes.
   - *Signature verifiability:* The verifiability of the resulting signature depends on the location of the fault in the subtree:
     - A *verifiable signature* is obtained when a fault hits any value involved in the authentication path of a non-top XMSS. Such an authentication path starts with the derivation of $2^{h'}-1$ W-OTS$^+$ public keys, as well as the computation of $2^{h'-i}-1$ nodes at each level $1 \leq i \leq h'$ of the subtree. Every W-OTS$^+$ public key requires the derivation of ℓ secret values; each of them chained $W-1$ times with the chaining pseudorandom function, so that all the results can be compressed with $\mathbf{T}_\ell$. This amounts to a total number of verifiable signatures of:

       $$\begin{array}{rcl}\#\mathrm{Verif}^X & = & (2^{h'}-1)(\ell + \ell(W-1)+1) + \sum_{i=1}^{h'}(2^{h'-i}-1) \\ & = & (2^{h'}-1)(\ell W+1) + 2^{h'} - h' - 1.\end{array}$$

     - A *non-verifiable but correct* signature is obtained when a fault hits any value on the path from a leaf to the root of a non-top XMSS. The values in a path consist of a single W-OTS$^+$ public key, in addition to a single node in all levels of a tree. As above, the W-OTS$^+$ public key requires the derivation of ℓ secret values; each of them chained $W-1$ times with the chaining pseudorandom function, so that all the results can be compressed with $\mathbf{T}_\ell$.

Since there are h' levels, this amounts to a total number of non-verifiable but correct signatures of:

$$\#\text{Non-verif}^X = \ell W + 1 + \sum_{i=1}^{h'} 1 = \ell W + 1 + h'.$$

5. **XMSS signature** at the top layer (i.e., $l^* = d$).
   - *Total hash function calls:* $\#\text{Total}^X = 2^{h'}(\ell W + 2) - 1$.
   - *Fault exploitability:* No.
   - *Signature verifiability:* The resulting signature is *non-verifiable but correct*, as the reconstruction of this XMSS does not lead to the SPHINCS$^+$ public key. All the W-OTS$^+$ signatures involved are valid, however no valid top part can be extracted.

Summing up the hash function calls of all the components above, the grand total of hash function calls in a single SPHINCS$^+$ signature is therefore given by:

$$\#\text{Total} = 1 + 1 + \#\text{Total}^F + d \cdot \#\text{Total}^X = 3 + k(3t - 1) + d(2^{h'}(\ell W + 2) - 1).$$

Table 3 computes the total numbers of possible verifiable and non-verifiable faulty signatures for both FORS and non-top XMSS in all SPHINCS$^+$ parameters sets. This table shows that a random fault is much likelier to give a verifiable signature rather than a non-verifiable one, and so that verifying the signature is not effective in detecting faulty signatures.

Table 3: Proportion of verifiable vs. non-verifiable signatures for faulty FORS and (non-top) XMSS for all SPHINCS$^+$ parameters sets.

	FORS ($l^* = 0$)				XMSS ($1 \leq l^* < d$)			
	Verif.		Non-verif.		Verif.		Non-verif.	
	Total	Ratio	Total	Ratio	Total	Ratio	Total	Ratio
128s	982,860	0.9998	171	0.0002	143,302	0.9960	569	0.0040
128f	45,720	0.9928	331	0.0072	3,931	0.8745	564	0.1255
192s	2,752,246	0.9999	253	0.0001	208,582	0.9961	825	0.0039
192f	24,981	0.9869	331	0.0131	5,723	0.8747	820	0.1253
256s	1,080,970	0.9997	353	0.0003	273,862	0.9961	1,081	0.0039
256f	91,770	0.9961	361	0.0039	16,106	0.9373	1,077	0.0627

Suppose that a fault can hit any hash function call uniformly at random. The above enumerations lead to the following probabilities:

Fault exploitability. The probability that the faulty signature is exploitable is given by the proportion of faulty signature outcomes that leads to an exploitable signature:

$$\mathbb{P}(\text{Expl.}) = \frac{\#\text{Total}^F + (d-1) \cdot \#\text{Total}^X}{\#\text{Total}} = \frac{k(3t-1) + 1 + (d-1)(2^{h'}(\ell W + 2) - 1)}{3 + k(3t-1) + d(2^{h'}(\ell W + 2) - 1)}.$$

Fault verifiability. Similarly, the probability that the faulty signature is verifiable is given by the proportion of the faulty signature outcomes that leads to a verifiable signature:

$$\begin{aligned}\mathbb{P}(\text{Verif.}) &= \frac{1 + \#\text{Verif}^F + (d-1) \cdot \#\text{Verif}^X}{\#\text{Total}} \\ &= \frac{1 + k(3t - a - 3) + (d-1)((2^{h'} - 1)(\ell W + 1) + 2^{h'} - h' - 1)}{3 + k(3t-1) + d(2^{h'}(\ell W + 2) - 1)}.\end{aligned}$$

Layer hit. Let $L = l^*$ denote the event that a fault has affected $\sigma_{l^*}^X$ (i.e., that a hash function call in the layer $l^* - 1$ is hit by a fault). The probability that $L = l^*$ is therefore given by the total number of hash function calls at layer $l^* - 1$:

$$\mathbb{P}(L = l^*) = \begin{cases} \dfrac{\#\mathrm{Total}^F}{\#\mathrm{Total}} = \dfrac{k(3t-1)+1}{3+k(3t-1)+d(2^{h'}(\ell W+2)-1)} & \text{if } l^* = 0, \\ \dfrac{\#\mathrm{Total}^X}{\#\mathrm{Total}} = \dfrac{2^{h'}(\ell W+2)-1}{3+k(3t-1)+d(2^{h'}(\ell W+2)-1)} & \text{if } 1 \le l^* \le d. \end{cases}$$

Table 4 computes the above probabilities given all SPHINCS$^+$ parameters sets. This table shows that the probability that a random fault leads to both an exploitable and a verifiable faulty signature is high.

Table 4: Fault analysis results for all SPHINCS$^+$ parameters.

	$\mathbb{P}$(Expl.)	$\mathbb{P}$(Verif.)		$\mathbb{P}(L = l^*)$				
			$l^* =$	0	1	...	$d-1$	d
128s	0.9326	0.9306		0.4607	0.0674	...	0.0674	0.0674
128f	0.9669	0.8857		0.3387	0.0331	...	0.0331	0.0331
192s	0.9527	0.9513		0.6216	0.0473	...	0.0473	0.0473
192f	0.9613	0.8576		0.1495	0.0387	...	0.0387	0.0387
256s	0.9162	0.9138		0.3296	0.0838	...	0.0838	0.0838
256f	0.9553	0.9095		0.2398	0.0447	...	0.0447	0.0447

4.2 Universal forgery analysis: one-fault model

This section analyzes the use case where the adversary has access to many valid signatures (i.e., $M_v > 1$) but only a single faulty one (i.e., $M_f = 1$) which is supposed exploitable and which corresponds to layer $0 \le l^* < d$. Let $N = 2^{h-h'l^*}$ be the total number of W-OTS$^+$ key pairs on layer l^*.

Collecting the corresponding valid signature. The probability that the valid signature corresponding to the same key pair as the faulty signature is included in the collected M_v signatures is simply given by:

$$\mathbb{P}(\text{W-OTS}^+ \text{ break}) = 1 - \left(1 - \frac{1}{N}\right)^{M_v}.$$

Alternatively, the expected number of valid queries to obtain the corresponding valid signature is given by a geometric random variable with probability $1/N$:

$$\mathbb{E}(M_v) = N.$$

Table 5 computes the expected numbers of valid queries to obtain in order to mount the universal forgery for each SPHINCS$^+$ parameters set. The average number of queries required to mount the forgery is in most cases lower than NIST's security definition for digital signatures of 2^{64} (see [NIS16]).

4.3 Universal forgery analysis: multiple-fault model

This section analyzes the use case where the adversary has access to multiple valid and faulty signatures (i.e., $M_v > 1$, $M_f > 1$) which are all supposed to be exploitable, different,

Table 5: Average numbers of valid signatures to collect the valid signature corresponding to a single faulty signature for all SPHINCS$^+$ parameters.

	$\mathbb{E}(M_v)$				
$l^* =$	0	1	...	$d-1$	d
128s	2^{64}	2^{56}	...	2^{8}	–
128f	2^{60}	2^{57}	...	2^{3}	–
192s	2^{64}	2^{56}	...	2^{8}	–
192f	2^{66}	2^{63}	...	2^{3}	–
256s	2^{64}	2^{56}	...	2^{8}	–
256f	2^{68}	2^{64}	...	2^{4}	–

and which all correspond to the same layer $0 \leq l^* < d$. Let $N = 2^{h-h'l^*}$ be the total number of W-OTS$^+$ key pairs on layer l^*. Also, let $\left\{ {a \atop b} \right\}$ denote the Stirling number of the second kind which counts the number of ways to distribute a objects into b non-empty subsets.

Faulty signatures collision. As the universal forgery can be mounted with only faulty signatures, the probability that two faulty signatures correspond to the same W-OTS$^+$ key pair is an instance of the birthday paradox [FGT92]:

$$\mathbb{P}(\text{W-OTS}^+ \text{ break}) = 1 - \frac{N!}{N^{M_f}(N - M_f)!}.$$

Pair of valid and faulty signatures. Combining the faulty signatures with the valid ones, the probability that a faulty signature corresponds to the same W-OTS$^+$ key pair as a valid signature is an instance of the occupancy problem with two types of balls [NS88]:

$$\mathbb{P}(\text{W-OTS}^+ \text{ break}) = 1 - \frac{1}{N^{M_v+M_f}} \sum_{t_v=1}^{M_v} \sum_{t_f=1}^{M_f} \left\{ {M_v \atop t_v} \right\} \left\{ {M_f \atop t_f} \right\} \frac{N!}{(N - M_v - M_f)!}.$$

Table 6 computes the above probabilities with $N = 256$ (i.e., when $l^* = d - 1$ for the 128s, 192s, and 256s parameters sets of SPHINCS$^+$, or $l^* = d - 2$ for SPHINCS$^+$-256f). This table shows that the randomness plays in the favor of the adversary, as only very few faulty queries are required to break a W-OTS$^+$. This number drops even lower when combined with very few valid queries.

Table 6: Probability of collision with either only faulty queries (under $M_v = 0$) or with M_f faulty and M_v valid queries ($N = 256$). Symmetrical values were omitted.

$M_f \backslash M_v$	0	4	8	16	32	64
4	0.0233	0.0607	0.1177	0.2215	0.3939	0.6325
8	0.1046		0.2215	0.3939	0.6325	0.8647
16	0.3803			0.6325	0.8647	0.9815
32	0.8676				0.9815	0.9996
64	0.9997					1.0000

Increasing the numbers of faulty signatures. While a single pair of different W-OTS$^+$ signatures corresponding to a same key pair is enough to mount the universal forgery, the grafting step becomes easier the more faulty W-OTS$^+$ signatures are obtained for a same key pair (see Section 3.3). In order to study this, notice that collecting M_f faulty signatures from N key pairs can be modeled as a multinomial distribution with uniform probabilities (i.e., $p_k = 1/N$ for $1 \leq k \leq N$).

The probability that at least one W-OTS$^+$ key pair has been reused c times is an instance of the maximal frequency in a multinomial distribution. Let s_k determine the accumulated number of W-OTS$^+$ signatures counting from the first W-OTS$^+$ key pair to the k^{th} key pair (so $s_0 = 0$ and $s_N = M_f$). Then, from the analysis by Corrado in [Cor11], the transition probability from s_{k-1} to s_k is given by:

$$\mathbb{P}(s_k \mid s_{k-1}) = \binom{M_f - s_{k-1}}{s_k - s_{k-1}} {\pi_k}^{s_k - s_{k-1}} (1-\pi_k)^{M_f - s_k},$$

where $\pi_k = p_k/(\sum_{i=k}^{N} p_k) = (1/N)/(\sum_{i=k}^{N} 1/N) = 1/(N-k)$.

Given the above probabilities, the stochastic matrix that determines the transitions from s_{k-1} to s_k is defined as follows:

$$\begin{aligned}
\mathbf{Q}_k &= \begin{pmatrix} \mathbb{P}(0 \mid 0) & \mathbb{P}(1 \mid 0) & \dots & \mathbb{P}(M_f \mid 0) \\ 0 & \mathbb{P}(1 \mid 1) & \dots & \mathbb{P}(M_f \mid 1) \\ \vdots & & \ddots & \vdots \\ 0 & \dots & & 1 \end{pmatrix} \quad \text{for } 1 \leq k \leq N-1, \\
\mathbf{Q}_N &= \begin{pmatrix} 1 & 1 & \dots & 1 \end{pmatrix}^\top.
\end{aligned}$$

Let $\bar{\mathbf{Q}}_k$ be the result of culling the transition probabilities that assign more than c signatures to a key pair (i.e., by setting $\mathbb{P}(s_k - s_{k-1} > C) = 0$ for the relevant s_k, s_{k-1}) and let $\bar{\mathbf{Q}}_1^{(1)}$ be the first row of $\bar{\mathbf{Q}}_1$.

The probability that the maximum load is no more than c is given by the transition from s_1 to s_n which is determined by the following product of stochastic matrices:

$$\mathbb{P}(\text{Max. load} \leq c \mid M_f) = \bar{\mathbf{Q}}_1^{(1)} \times \bar{\mathbf{Q}}_2 \times \ldots \times \bar{\mathbf{Q}}_N.$$

Alternatively, the expected maximum load given M_f faulty signatures is:

$$\mathbb{E}(\text{Max. load} \mid M_f) = \sum_{c=0}^{M_f - 1} \mathbb{P}(\text{Max. load} > c \mid M_f).$$

Combining this result with the valid signatures, notice that the maximum load is increased by one by collecting the valid signature of the W-OTS$^+$ for which the maximum load is reached. As such an event can be modeled as a Bernoulli random variable with probability $1 - (1 - 1/N)^{M_v}$, we ultimately have:

$$\mathbb{E}(\text{Max. load} \mid M_v, M_f) = \mathbb{E}(\text{Max. load} \mid M_f) + \left(1 - \left(\frac{N-1}{N}\right)^{M_v}\right).$$

Table 7 computes the maximum load averages with M_f signatures for the N that correspond to the few first top layers of the SPHINCS$^+$ parameters sets, as increasing the number of signatures is especially relevant when targeting such layers.

Layer coverage. The probability that the collected valid signatures cover the entire layer—in which case, all valid signatures are known—is an instance of the coupon collector's problem [FGT92]:

$$\mathbb{P}(\text{Layer is covered}) = \frac{N!}{N^{M_v}} \left\{ \begin{matrix} M_v - 1 \\ N - 1 \end{matrix} \right\}.$$

Table 7: Maximum load averages with various numbers of faulty signatures M_f in different layers of N signatures.

$N \backslash M_f$	64	128	256	512	1,024
2^3	12.23	21.90	40.26	75.60	144.31
2^4	7.88	13.35	23.43	42.36	78.50
2^6	3.96	5.97	9.37	15.33	26.10
2^8	2.46	3.38	4.77	6.99	10.69
2^9	2.12	2.74	3.68	5.16	7.48

Alternatively, the expected number of valid queries to cover the entire layer is given by:

$$\mathbb{E}(M_v \text{ to cover layer}) = N \sum_{i=1}^{N} \frac{1}{i}, \quad \text{which is } \Theta(N \log N).$$

Table 8 computes the expected numbers of queries required to cover the few first top layers of the SPHINCS$^+$ parameters sets, as obtaining all valid signatures are especially practical when targeting such layers.

Table 8: Average numbers of valid signatures to cover various layers of N signatures.

N	2^3	2^4	2^6	2^8	2^9	2^{12}	2^{16}
$\mathbb{E}(M_v)$	$2^{4.44}$	$2^{5.76}$	$2^{8.25}$	$2^{10.61}$	$2^{11.77}$	$2^{15.15}$	$2^{19.54}$

5 Caching countermeasures analysis

In order to prevent faulty signatures from being collected, the W-OTS$^+$ signatures computed throughout a SPHINCS$^+$ signing procedure can be cached (i.e., stored in memory, sometimes temporarily, and then retransmitted without recomputation when requested). Such a process not only prevents accidental faulty recomputations of a W-OTS$^+$ signature, but also improves the performances of the SPHINCS$^+$ signature generation. Notice also that the valid W-OTS$^+$ signatures are leakage-agnostic, so the cache can therefore be shared with verifiers (in a read-only fashion).

In this section, we consider two different strategies of caching W-OTS$^+$s: caching layers and caching branches.

5.1 Caching layers

This strategy, originally proposed in Gravity-SPHINCS [AE17], consists of caching all the W-OTS$^+$ within one or more layers (starting from the top layer). Since the cache is not updated with new signature requests, the cache is static and can therefore be added to the public key.

Algorithms. Let c be the number of layers for which all W-OTS$^+$ signatures and public keys are cached. The countermeasure consists of changing the key generation algorithm and the signing procedure algorithm of the hypertree and XMSS with the following:

- The new *key generation* procedure consists of discovering all the XMSSs from layers $d-1-c$ to $d-1$ and storing all the W-OTS$^+$ signatures and public keys on the way to the top XMSS. The secret and public keys are the same.

  This strategy increases the complexity of the key generation by a factor of $\sum_{i=0}^{c} 2^{h'i} = (2^{ch'+h'} - 1)/(2^{h'} - 1)$.

- The new *signing procedure* derives the XMSS signatures for the cached layers by using the W-OTS$^+$ signatures and public keys from the cache. An n-byte digest `msg` at hyperleaf index $1 \leq \lambda \leq 2^h$ is therefore signed as follows:

  1. For $0 \leq i < d - c$:
     (a) Derive τ_i, λ_i from τ_{i-1} (starting with $\tau_{-1} = \lambda$).
     (b) Generate the XMSS key pair $(\textsc{sk}_i^X, \textsc{pk}_i^X)$ at the address corresponding to τ_i.
     (c) Sign r with $\textsc{sk}^X$ using λ_i as leaf index to produce σ_i and update r with $\textsc{pk}_i^X$.
  2. For $d - c \leq i < d$:
     (a) Derive τ_i, λ_i from τ_{i-1} (starting with $\tau_{-1} = \lambda$).
     (b) Read σ_i^W from the cache at tree index τ_i and leaf index λ_i.
     (c) Compute the XMSS authentication path auth$_i$ starting from the leaf λ_i and using, as leaves, the cached W-OTS$^+$ public keys at tree index τ_i.
     (d) Let $\sigma_i = (\sigma_i^W, \text{auth}_i)$.
  3. Return $\sigma^{HT} = (\sigma_0, \ldots, \sigma_{d-1})$.

  This strategy decreases the signing procedure complexity of $c \times 2^{h'}(\ell W + 1)$ hash function calls.

Analysis. While the algorithm prevents faulting c W-OTS$^+$ signatures, the new algorithm features a reduced total number of hash function calls which therefore impacts the proportion of vulnerable hash function calls, hence the chance that a random fault produces an exploitable faulty signature.

In a cached XMSS, the total number of hash function calls is: $\#\text{Total}^{\tilde{X}} = 2^{h'-1} - 1$. This leads to a new grand total of hash function calls in the SPHINCS$^+$ signing procedure:

$$\begin{aligned} \#\text{Total} &= 2 + \#\text{Total}^F + (d-c) \cdot \#\text{Total}^X + c \cdot \#\text{Total}^{\tilde{X}} \\ &= 3 + k(3t-1) + (d-c)(2^{h'}(\ell W + 2) - 1) + c(2^{h'} - 1). \end{aligned}$$

As a result, since a fault in an XMSS below a cached layer is not exploitable anymore, the proportion of hash function calls that lead to an exploitable faulty signature is:

$$\begin{aligned} \mathbb{P}(\text{Expl.}) &= \frac{\#\text{Total}^F + (d-c-1) \cdot \#\text{Total}^X}{\#\text{Total}} \\ &= \frac{1 + k(3t-1) + (d-c-1)(2^{h'}(\ell W + 2) - 1)}{3 + k(3t-1) + (d-c)(2^{h'}(\ell W + 2) - 1) + c(2^{h'} - 1)} \end{aligned}$$

where $0 < c < d$ ($\mathbb{P}(\text{Expl.}) = 0$ if $c = d$).

Table 9 shows how the probability that a single random fault is exploitable decreases with c for all SPHINCS$^+$ parameter sets. This table shows that the probability that a random fault gives an exploitable faulty signature stays fairly high, especially for the fast variants of SPHINCS$^+$.

In terms of memory, let C denote the total number of W-OTS$^+$ signatures cached. We therefore obtain:

$$C = \sum_{i=1}^{c} 2^{h'i} = 2^{h'}(2^{ch'} - 1)/(2^{h'} - 1).$$

As a W-OTS$^+$ signature consists of ℓ elements of n bytes and since a W-OTS$^+$ public key consists of a single element of n bytes, caching c layers requires $C(\ell + 1)n$ bytes in total. Table 10 shows how the cost of caching layers evolves with c for all SPHINCS$^+$ parameter sets. This table demonstrates that the memory requirements for this countermeasure blows up very early and that only the first few top layers can be cached in practice.

Table 9: Analysis of the layer caching countermeasure for all SPHINCS$^+$ parameter sets.

	$\mathbb{P}$(Expl.)						
$c =$	1	2	3	4	…	$d-1$	d
128s	0.8972	0.8591	0.8179	0.7733	…	0.6141	0.0000
128f	0.9505	0.9335	0.9158	0.8975	…	0.5076	0.0000
192s	0.9287	0.9034	0.8767	0.8486	…	0.7539	0.0000
192f	0.9420	0.9218	0.9007	0.8787	…	0.2625	0.0000
256s	0.8711	0.8216	0.7670	0.7066	…	0.4784	0.0000
256f	0.9327	0.9090	0.8840	0.8578	…	0.3864	0.0000

Table 10: Analysis of the layer caching countermeasure for all SPHINCS$^+$ parameter sets.

	Memory (bytes)					
$c =$	1	2	3	4	…	d
128s	1.43×10^5	3.68×10^7	9.43×10^9	2.41×10^{12}	…	1.04×10^{22}
128f	4.48×10^3	4.03×10^4	3.27×10^5	2.62×10^6	…	7.38×10^{20}
192s	3.13×10^5	8.05×10^7	2.06×10^{10}	5.28×10^{12}	…	2.27×10^{22}
192f	9.79×10^3	8.81×10^4	7.15×10^5	5.73×10^6	…	1.03×10^{23}
256s	5.49×10^5	1.41×10^8	3.61×10^{10}	9.24×10^{12}	…	3.97×10^{22}
256f	3.43×10^4	5.83×10^5	9.36×10^6	1.50×10^8	…	6.75×10^{23}

5.2 Caching branches

This strategy consists of caching all the W-OTS$^+$ signatures and public keys in a path during a signing procedure. The cache is dynamic and may require to be updated for each new signature.

As reported in [GKPM18], this strategy completely prevents similar fault-based universal forgeries in stateful hash-based signature schemes (such as XMSS^{MT} [HRB13]). This is because the subtrees involved in stateful schemes provide only a limited number of signatures whose availability is remembered by the signer. Thus, once computed, the signature of a subtree can be retained as long as the subtree is involved in new signatures, at which point it is replaced by the next subtree in line. This prevents faulty recomputations of the signatures by caching only one W-OTS$^+$ per layer. This section shows that applying the same idea to SPHINCS$^+$ is ineffective, even when multiple W-OTS$^+$s per layer are cached.

Algorithms. The countermeasure consists of adding a cache of size C_l to each layer $0 \le l < d$ of XMSSs, where $C_l \le 2^{h'l}$ denotes the number of W-OTS$^+$ signatures and public keys that can be stored in the cache at layer l. The new XMSS signing procedure therefore signs an n-byte digest `msg` at leaf index $1 \le \lambda \le 2^{h'}$ as follows:

1. Check if the W-OTS$^+$ signature at leaf index λ is in the cache:
    - On *cache hit*, read the signature σ^W_λ and $\textsc{pk}^W_\lambda$ from the cache.
    - On *cache miss*:
        (a) If the cache is full, evict the least recent signature.
        (b) Use $\textsc{sk}^X_\lambda$ to produce $\textsc{pk}^W_\lambda$ and σ^W_λ; the W-OTS$^+$ signature of `msg`.
        (c) Put σ^W_λ and $\textsc{pk}^W_\lambda$ in the cache.
2. Compute the authentication path auth_λ starting from $\textsc{pk}^W_\lambda$ (using cached W-OTS$^+$ public keys when accessible).
3. Return $\sigma^X = (\sigma^W_\lambda, \text{auth}_\lambda)$.

When all caches are filled, this strategy enhances the XMSS signing procedure complexity of an average of $\sum_{l=0}^{d-1} 2^{h'}(\ell W + 1)(C_l/2^{h-h'l})$ fewer hash function calls. We suppose all caches empty at the device startup.

Analysis. As not all branches of the hypertree can realistically be cached, in order for the countermeasure to be effective, we suppose that we cache only a significant ratio of a layer. We furthermore focus on the significantly cached layer, as the layers above are necessarily all cached while the layers below are only marginally covered.

A faulty signature is exploitable if the fault hits a layer for which the corresponding W-OTS$^+$ signature is uncached. As the cache is dynamically filled, the probability of a cache miss depends on the number of distinct signatures visited after M queries to the signing procedure.

Let D_l denote the number of distinct visited W-OTS$^+$ signatures in layer $0 \leq l < d$ after M queries, and $N = 2^{h-h'l}$ the total number of W-OTS$^+$ signatures in such layer. Then, the distribution of D_l is an instance of the occupancy problem [Fel67]:

$$\mathbb{P}(D_l = i) = \frac{N!\alpha_{i,M}}{(N-i)!N^M}, \quad \text{where } \alpha_{i,M} = \frac{1}{i!}\sum_{k=1}^{i}(-1)^{i-k}\binom{i}{k}k^M \quad (1 \leq i \leq N).$$

Now, suppose that a total of $D_l \leq 2^{h-h'l}$ signatures are cached at each layer $0 \leq l < d$ (after a certain number M of queries). Then, as before, the probability that a fault leads to an exploitable faulty W-OTS$^+$ signature is derived by counting the number of vulnerable hash function calls in the procedure. However, in this case, the totals of hash function calls at all layers behave as random variables which depend on the cache status of every leaf in each XMSS. So, instead of deriving the exact totals, we evaluate the following heuristic:

$$\mathbb{P}(\text{Expl.}) = \frac{\mathbb{E}(\#\text{Expl.})}{\mathbb{E}(\#\text{Total})}$$

where $\mathbb{E}(\#\text{Expl.})$ denotes the average number of hash function calls that lead to an exploitable faulty signature when faulted, and $\mathbb{E}(\#\text{Total})$ the average total number of hash function calls in a SPHINCS$^+$ signing procedure.

Starting with the average total of hash function calls, notice that only the XMSS signing procedure was changed. Supposing that the cache is uniformly filled, we obtain:

$$\begin{aligned}
\mathbb{E}(\#\text{Total}) &= 2 + \#\text{Total}^F + \sum_{l=0}^{d-1} \mathbb{E}\left(\#\text{Total}^{\tilde{X}_l}\right) \\
&= 2 + \#\text{Total}^F + \sum_{l=0}^{d-1}\left(2^{h'} - 1 + \sum_{i=1}^{2^{h'}} \mathbb{P}(\text{Cache miss at layer } l)(\ell W + 1)\right) \\
&= 2 + \#\text{Total}^F + \sum_{l=0}^{d-1}\left(2^{h'} - 1 + 2^{h'}\left(1 - \frac{D_l}{2^{h-h'l}}\right)(\ell W + 1)\right).
\end{aligned}$$

The average total of vulnerable hash function calls is determined by the average total number of hash function calls in each layer of the SPHINCS$^+$ structure. At each layer, such a number now depends on the number of W-OTS$^+$ cached on the layer, as well as the number of W-OTS$^+$ cached on the layer above. Again, supposing that the cache is uniformly distributed, we obtain:

$$\begin{aligned}
\mathbb{E}(\#\text{Expl.}) &= \mathbb{E}\left(\#\text{Expl.}^{\tilde{F}}\right) + \sum_{l=0}^{d-2} \mathbb{E}\left(\#\text{Expl.}^{\tilde{X}}\right) \\
&= \mathbb{P}(\text{Cache miss at layer } 0) \cdot \#\text{Total}^{F} + \\
&\qquad \sum_{l=0}^{d-2} \mathbb{P}(\text{Cache miss at layer } l+1) \cdot \mathbb{E}\left(\#\text{Total}^{\tilde{X}}\right) \\
&= \left(1 - \frac{D_{2^h}}{2^h}\right)(3 + k(3t-1)) + \\
&\qquad \sum_{l=0}^{d-2} \left(1 - \frac{D_{l+1}}{2^{h-h'(l+1)}}\right)\left(2^{h'} - 1 + 2^{h'}\left(1 - \frac{D_l}{2^{h-h'l}}\right)(\ell W + 1)\right).
\end{aligned}$$

Table 11 shows how the probability that a random fault is exploitable decreases with b for all SPHINCS$^+$ parameter sets supposing that all the caches are filled to capacity (i.e., $D_l = \min(b, 2^{h-h'l})$, so after a sufficiently large number of queries M were made). As with caching layers, since the total number of hash function calls in the entire signing procedure decreases with the number of vulnerable hash function calls, the proportion of exploitable faulty signatures stays fairly high, especially for the fast variants of SPHINCS$^+$. Note however that such a countermeasure still leaves fewer hash function calls vulnerable than an unprotected SPHINCS$^+$.

Table 11: Analysis of the branch caching countermeasure for all SPHINCS$^+$ parameter sets. The numbers b are rounded up to the next integer.

	$\mathbb{P}(\text{Expl.})$					
$b =$	$(2/3)2^{h'}$	$(2/3)2^{2h'}$	$(2/3)2^{3h'}$	$(2/3)2^{4h'}$	…	$(2/3)2^{dh'}$
128s	0.9292	0.9238	0.9174	0.9098	…	0.3172
128f	0.9647	0.9634	0.9620	0.9605	…	0.3219
192s	0.9511	0.9485	0.9457	0.9425	…	0.3249
192f	0.9585	0.9568	0.9549	0.9528	…	0.3052
256s	0.9111	0.9023	0.8917	0.8785	…	0.3068
256f	0.9530	0.9507	0.9481	0.9453	…	0.3130

Since the universal forgery requires at least two recomputations of the same W-OTS$^+$ signature, we study the number of queries before a W-OTS$^+$ signature needs to be recomputed (i.e., two cache misses for a same W-OTS$^+$). We solve this problem with a Markov chain (see, e.g., [GS97] for a reference on the methodology) as shown in Figure 6. The corresponding transition matrix $\mathbf{P} = \left(p_{i,j}\right)$ is defined as follows (for $0 \leq i, j \leq N+2$):

$$p_{i,j} = \begin{cases} \min(i, C_l)/N & \text{if } j = i \neq N+1, \\ (N-i)/N & \text{if } j = i+1, \\ (i - C_l)/N & \text{if } j = N+1 \text{ and } i > C_l, \\ 1 & \text{if } j = i = N+1, \\ 0 & \text{otherwise.} \end{cases}$$

The fundamental matrix that counts the average number of discrete steps spent in each state is computed as follows:

$$\mathbf{N} = (\mathbf{I} - \mathbf{Q})^{-1},$$

where $\mathbf{I}$ is the $(N+1) \times (N+1)$ identity matrix, and $\mathbf{Q}$ is the $(N+1) \times (N+1)$ submatrix of $\mathbf{P}$ without the last column and row. As we start with the cache being empty, the

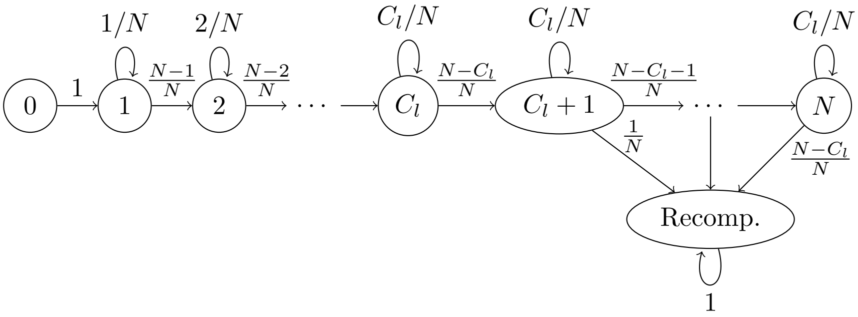


Figure 6: Markov chain representing the transitions from the cache being empty to any W-OTS$^+$ being recomputed. The states (others than "Recomp.") count the number of cache misses without recomputation.

expected number of queries M before a W-OTS$^+$ is recomputed is given by summing the first row of the fundamental matrix:

$$\mathbb{E}(M \text{ to recomp.}) = \sum_{j=0}^{N} \mathbf{N}_{0,j}.$$

Table 12 computes the expected numbers of queries required so that any W-OTS$^+$ gets recomputed for few first layers of all the SPHINCS$^+$ parameter sets, given various cache sizes. This table shows that the recomputation of a W-OTS$^+$ signature can be triggered with very few queries.

Table 12: Average number of queries such that a W-OTS$^+$ is recomputed for various cache sizes C_l and different layers of N signatures.

$N \backslash C_l$	$(1/2)N$	$(2/3)N$	$(3/4)N$	$N-1$
2^3	$2^{3.53}$	$2^{4.30}$	$2^{4.30}$	$2^{4.89}$
2^4	$2^{4.26}$	$2^{4.85}$	$2^{5.07}$	$2^{6.13}$
2^6	$2^{5.89}$	$2^{6.49}$	$2^{6.78}$	$2^{8.52}$
2^8	$2^{7.69}$	$2^{8.31}$	$2^{8.63}$	$2^{10.83}$
2^9	$2^{8.63}$	$2^{9.26}$	$2^{9.58}$	$2^{11.97}$

In terms of memory, let C denote the total number of W-OTS$^+$ signatures cached when b branches are fully cached. As $C_l \leq 2^{h'l}$, we have that:

$$C = \sum_{l=0}^{d-1} \min(b, 2^{h-h'l}).$$

As with caching layers, a W-OTS$^+$ signature consists of ℓ elements of n bytes and a W-OTS$^+$ public key consists of a single element of n bytes, so caching b layers requires $C(\ell+1)n$ bytes in total. Table 13 shows the cost of caching various numbers of branches for all SPHINCS$^+$ parameter sets. The memory requirements for this countermeasure blows up very early, so only the first few top layers are expected to be covered in practice.

6 Practical experiments

The following section aims to experimentally verify the fault attack as described in Section 3, the analysis of the fault attack from Section 4, and the analysis of the caching countermeasures from Section 5.

Table 13: Analysis of the branch caching countermeasure for all SPHINCS$^+$ parameter sets. The numbers b are rounded up to the next integer.

	Memory (bytes)					
$b =$	$(2/3)2^{h'}$	$(2/3)2^{2h'}$	$(2/3)2^{3h'}$	$(2/3)2^{4h'}$	$\dots$	$(2/3)2^{dh'}$
128s	8.14×10^5	1.82×10^8	4.00×10^{10}	8.53×10^{12}	$\dots$	7.36×10^{21}
128f	7.14×10^4	4.91×10^5	3.71×10^6	2.80×10^7	$\dots$	5.55×10^{20}
192s	1.74×10^6	3.90×10^8	8.56×10^{10}	1.83×10^{13}	$\dots$	1.58×10^{22}
192f	1.68×10^5	1.16×10^6	8.81×10^6	6.69×10^7	$\dots$	7.62×10^{22}
256s	3.02×10^6	6.77×10^8	1.49×10^{11}	3.17×10^{13}	$\dots$	2.74×10^{22}
256f	4.13×10^5	6.08×10^6	9.12×10^7	1.36×10^9	$\dots$	4.79×10^{23}

6.1 Setup

Hardware. As the fault attack does not require sophisticated glitching technology, our proof of concept uses the ChipWhisperer framework to perform experiments, which includes:

- The Chipwhisperer-Lite Level 2 starter kit.
- A NAE-CW308T-STM32F4 as the Device Under Testing (DUT).
- A common laptop running linux 5.18.6-arch1-1.

The DUT is configured to run at its maximal clock frequency (i.e., 180 MHz).

Software. We attack the reference implementation of SPHINCS$^+$ from [FKN$^+$22] which was slightly adapted to run on the Cortex-M4 of the DUT. The instance attacked is `sphincs-shake-256s-robust` which is claimed to achieve the maximal theoretical security guarantees. The hash function SHAKE was instantiated with a portable software implementation.

For practicality purpose, the software was further modified to limit the signing procedure to the computation of a single layer. As a result, the software would use the W-OTS$^+$ keypair of an XMSS at a fixed layer $0 < l^* < d$ to sign the XMSS root at layer $l^* - 1$ addressed by a given index. The output signature consists of the W-OTS$^+$ signature along with the authentication path in the XMSS of layer $l^* - 1$.

The laptop communicates with the DUT through UART and the protocol is implemented using ChipWhisperer's simpleserial library. The DUT can be commanded to:

- Program the SPHINCS$^+$ secret and public seeds $\mathtt{sk}_1$ and $\mathtt{pk}_2$.
- Given an address, compute the W-OTS$^+$ signature and the authentication path of the XMSS at layer $l^* - 1$.
- Retrieve the bytes of the last W-OTS$^+$ signature and authentication path computed.

See `https://github.com/AymericGenet/SPHINCSplus-FA` for the source code.

Fault injection. To collect faulty signatures, the ChipWhisperer is used to inject a glitch in the system clock of the DUT. We do not synchronize the glitch injection with a trigger signal as we do not require to hit a precise instruction to collect exploitable signatures. Instead, the glitch is manually injected after a (progressive) delay that follows the communication with the DUT.

The glitch characteristics were explored experimentally to favor faulty signatures. Using a width of 20 samples and a clock offset of -4 samples, we report $\approx 1/3$ of output signatures to be faulty (so $\approx 2/3$ of valid outputs).

6.2 Experiment 1: randomized + cached layer

In the first experiment, we simulate the layer caching countermeasure (see Section 5) by pretending that all the W-OTS$^+$ signatures on the last layer are cached. In practice, such a cache would amount to 0.55 MB of ROM. The experiment therefore aims to show the feasibility of an attack on the second last layer (i.e., $l^* = d - 2 = 6$).

The experiment protocol to query a signature goes as follows:

1. The laptop sends to the DUT three bytes that correspond to the XMSS address at layer $l^* - 1$, i.e.:
   - $\tau_{l^*-1} =$ the first two bytes sent.
   - $\lambda_{l^*-1} =$ the last byte sent.
2. The DUT computes:
   (a) The authentication path of the XMSS at layer $l^* - 1$ and tree index τ_{l^*-1}, starting from the leaf index λ_{l^*-1}.
   (b) The root r of the XMSS at layer $l^* - 1$ and tree index τ_{l^*-1}.
   (c) The W-OTS$^+$ signature of r, using the W-OTS$^+$ key pair from the XMSS at layer l^*, tree index τ_{l^*}, and leaf index λ_{l^*}, where:
      - $\tau_{l^*} =$ the first byte of τ_{l^*-1}.
      - $\lambda_{l^*} =$ the last byte of τ_{l^*-1}.
3. The laptop then retrieves the W-OTS$^+$ signature and authentication path.

The DUT takes around 79 seconds to compute a single XMSS authentication path and W-OTS$^+$ signature, during which the clock glitch is blindly injected. We conduct $N = 5$ trials where a single trial consists of repeating the above with a fixed SPHINCS$^+$ secret seed to collect 1,024 potentially faulty signatures.

Results. The faulty signature collection is successful across all trials, as a W-OTS$^+$ is always found to be compromised at the end of the collection. Table 14 reports the types of signatures collected during the trials which were identified by recomputing the correct W-OTS$^+$ signature and authentication path from the programmed secret seed.

Table 14: Analysis of the collected signatures in $N = 5$ fault attacks against SPHINCS$^+$-shake-256s-robust at layer $l^* = 6$.

	Signatures		Faulty signatures		Non-verif. signatures	
	Valid	Faulty	Verif.	Non-verif.	Correct	Incorrect
Mean	660.4	363.6	269.6	94	1.8	92.2
SD	14.8762	14.8762	14.8257	11.2694	1.3038	10.1094
Min.	639	346	257	83	1	82
Max.	678	385	295	113	4	109

Table 15 reports the results related to the universal forgery. Given the analysis from Section 4 and using $M_f \approx (1/3)1024$ and $M_v \approx (2/3)1024$, we have that a W-OTS$^+$ signature is compromised with a probability of 0.5877 using only faulty signatures, and of 0.9714 using both valid and faulty signatures. The maximum load is expected to be $1.59 + 0.01$. On average, the probability that the grafting step is successful with two different W-OTS$^+$ signatures is $2^{-34.85}$. All these numbers are aligned with the ones obtained in practice.

Conclusion. The experiment has demonstrated that despite the fact that the layer presents 2^{16} signatures, as few as 2^{10} signature queries with a fault probability of $\approx 1/3$ are enough to compromise at least one W-OTS$^+$ and, therefore, mount a SPHINCS$^+$ universal forgery.

Table 15: Analysis of the universal forgery in $N = 5$ fault attacks against SPHINCS$^+$-shake-256s-robust at layer $l^* = 6$.

	Compromised W-OTSs$^+$	Maximum load	Best $\mathbb{P}$(grafting)
Mean	2.2	2	$2^{-35.7388}$
SD	1.7889	0	$2^{-35.5089}$
Min.	1	2	$2^{-47.0379}$
Max.	5	2	$2^{-34.2432}$

6.3 Experiment 2: randomized + cached branches

In the second experiment, we simulate the branch caching countermeasure (see Section 5) by implementing an internal cache of C addresses for which we pretend that the corresponding W-OTS$^+$ are transmitted without recomputation. When requesting a W-OTS$^+$ at a certain address, the computation is triggered only if the given address was not previously cached.

The experiment aims to show that an attack is possible even when a significant portion of the layer is cached. For practicality purpose, we target the last layer (i.e., $l^* = d-1 = 7$) and use a cache of size $C = 171$ to cover two thirds of the $2^{h'} = 256$ possible addresses. In theory, such a cache would amount to 2.93 MB of RAM.

At the beginning of the experiment, the DUT's cache is empty. The experiment protocol to query a signature goes as follows:

1. The laptop sends to the DUT two bytes that correspond to the XMSS address at layer $l^* - 1$, i.e.:
   - $\tau_{l^*-1} =$ the first byte sent.
   - $\lambda_{l^*-1} =$ the last byte sent.
2. If τ_{l^*-1} is in the DUT's cache, then the DUT computes nothing and the protocol stops here.
3. Else, if τ_{l^*-1} is not cached, then the DUT saves τ_{l^*-1} in the cache (after evicting the least recent address cached if the cache is full), and computes:
   (a) The authentication path of the XMSS at layer l^*-1 at tree index τ_{l^*-1}, starting from the leaf index λ_{l^*-1}.
   (b) The root r of the XMSS at layer $l^* - 1$ and tree index τ_{l^*-1}.
   (c) The W-OTS$^+$ signature of r, using the W-OTS$^+$ key pair from the XMSS at layer l^*, tree index τ_{l^*}, and leaf index λ_{l^*}, where:
   - $\tau_{l^*} = 0$.
   - $\lambda_{l^*} = \tau_{l^*-1}$.
4. The laptop then retrieves the W-OTS$^+$ signature and authentication path.

On a cache miss, the DUT takes around 79 seconds to compute a single XMSS authentication path and W-OTS$^+$ signature, during which the clock glitch is blindly injected. The glitch is not injected on a cache hit. We conduct a total of $N = 10$ trials where a single trial consists of repeating the above with a fixed SPHINCS$^+$ secret seed to collect 512 potentially faulty signatures.

Results. The faulty signature collection is successful across all trials, as a W-OTS$^+$ is always found to be compromised at the end of the collection. Table 16 reports the types of signatures collected during the trials which were identified by recomputing the correct W-OTS$^+$ signature and authentication path using the programmed secret seed.

Table 16: Analysis of the collected signatures in $N = 10$ fault attacks against SPHINCS$^+$-shake-256s-robust at layer $l^* = 7$ when 171 branches are cached.

	Signatures		Faulty signatures		Non-verif. signatures	
	Valid	Faulty	Verif.	Non-verif.	Correct	Incorrect
Mean	419.3	92.7	76.4	16.3	0.2	16.1
SD	7.4841	7.4841	5.1251	4.7854	0.42	4.7714
Min.	409	81	67	9	0	9
Max.	431	103	84	25	1	25

Table 17 reports the results related to the universal forgery. Given the analysis from Section 5, the number of queries before a W-OTS$^+$ is recomputed is 318.09. Using a probability of successful fault injection of 1/3, a W-OTS$^+$ is successfully compromised upon recomputation with a probability of $1 - (1 - 1/3)^2 = 0.5555$. This number is aligned with the ones obtained in practice.

Table 17: Analysis of the universal forgery in $N = 10$ fault attacks at layer $l^* = 7$ against SPHINCS$^+$-shake-256s-robust when 171 branches are cached.

	Queries before first recomp.	Compromised W-OTSs$^+$	Maximum load	Best $\mathbb{P}$(grafting)
Mean	318.6	14.1	2	$2^{-30.4274}$
SD	25.3693	3.7253	0	$2^{-30.3094}$
Min.	284	7	2	$2^{-35.7972}$
Max.	374	20	2	$2^{-28.8953}$

Conclusion. The experiment has demonstrated that despite the fact that two thirds of the attacked layer are cached, as few as 2^9 signature queries with a fault probability of $\approx 1/3$ are enough to compromise at least one W-OTS$^+$ and, thus, mount a SPHINCS$^+$ universal forgery.

7 Discussion & Conclusion

In this paper, a refined fault attack against SPHINCS$^+$ that is less restrictive than the original attack from Castelnovi, Martinelli, and Prest in [CMP18] has been presented. The complexity of the attack in terms of required queries, hashes, and success probability has also been scrupulously analyzed. Finally, the effectiveness of countermeasures based on caching both layers and branches has been shown to be underwhelming; a result which was experimentally verified.

The main takeaway of the current analysis is that SPHINCS$^+$ is extremely fragile against faults. As Section 4 shows, a *single* unconstrained corruption of *almost any* computation has a catastrophic impact on the security guarantees of *all* SPHINCS$^+$ parameters sets. This amounts to *millions* of hash function calls that need to be carried out faultlessly in order to sign a single message; a number that is not considering other subroutines (such as, e.g., the checksum in W-OTS$^+$) which are at least equally vulnerable.

While the other post-quantum signature algorithms selected by NIST in 2022 are also susceptible to fault attacks, this vulnerability makes SPHINCS$^+$ the most sensitive candidate to faults. For example, Bruinderink and Pessl have demonstrated in [BP18] that the lattice-based signature scheme CRYSTALS-Dilithium is also vulnerable to a universal forgery using an equivalent fault model. However, the attack on CRYSTALS-Dilithium can only be mounted when an adversary obtains the valid and faulty signatures of the same message, while SPHINCS$^+$ is vulnerable even when the device signs different and uncontrolled messages. Additionally, while the authors of [BP18] suggest that verifying signatures or randomization can serve as effective countermeasures against differential fault attacks on CRYSTALS-Dilithium, both of these approaches have been shown to be ineffective when applied to SPHINCS$^+$. The current attacks against Falcon—another lattice-based signature scheme chosen by NIST—only work when these faults result in an early abortion and zeroing of values, which requires a higher precision and more capabilities than the fault model considered in this paper (see [MHS$^+$19]).

Such a fragility needs to be taken seriously, as faults are reported to naturally happen in conventional hardware such as, e.g., in DRAM. For instance, Schroeder, Pinheiro, and Weber have reported 25,000 to 70,000 errors in DRAM per billion device hours per MBit in Google's 2009 fleet [SPW11]. As a result, with long enough deployments of SPHINCS$^+$ on standard computers, the fault attack is eventually going to affect real-world users.

As ordinary hardware cannot be fully trusted to protect against faults, and since faults can also be maliciously injected, a proper countermeasure that entirely prevents the fault attack is preferable. However, the problem is not obvious to solve, as the universal forgery exploits the fact that the signing procedure recomputes one-time signatures; a core feature of the SPHINCS family that makes the scheme practical and stateless. Yet, as long as one-time signatures are being recomputed on the fly, the risk of reusing a one-time key pair to sign an unexpected message will always be present (which, in practice, is accomplished with a fault injection). While this problem is solved in stateful schemes such as XMSS^{MT} by caching the relevant one-time signatures, Section 5 shows that the same countermeasure fails to properly protect SPHINCS$^+$.

Since the threat of a fault can never be completely eliminated, the current best solution to protect the signature scheme against accidental and intentional faults is through *redundancy*; an observation that is shared by others (see [CMP18, ALCZ20]). Redundancy consists of recomputing a same signature multiple times (ideally, with different implementations) and abort the procedure in case a mismatch is detected. Even though parallelizable, this solution at least doubles the signing time which strikes a huge blow to the performance of the scheme which was already lacking in the original submission. Specially protected implementations on the hardware level, as recommended in the SPHINCS$^+$ specifications [HBD$^+$20], may also offer an adequate protection against faults but would require fault-protection mechanisms not only in the hash function implementation, but also in the other subroutines of the scheme, as well as in the device memory.

In conclusion, the results of this paper urge all real-world deployments of SPHINCS$^+$ to come with redundancy checks, even if the use case is not prone to faults (such as, e.g., with firmware updates). Unless an adversary can query the signature for any message, randomized signing may be disabled as such measure is not a reliable way to prevent the fault attack. Verification, on the other hand, is still recommended as non-verifiability (even though unlikely) implies the occurrence of a fault.

Future work. The results of this paper call for novel countermeasures that make SPHINCS$^+$ inherently resistant to fault attacks. As argued above, such a solution should avoid the accidental or intentional recomputation of one-time signatures which will likely necessitate a new way of performing hash-based signatures. For instance, an ambitious reader might come up with a solution that changes the one-time signatures in SPHINCS$^+$ by one-*message* signatures which, if such a primitive makes sense, might even lead to an entirely new scheme. Other solutions that, for instance, make faulty signatures always non-verifiable would also be a desirable step forward, so a signing device could at least block bad signatures by running the verification procedure on the produced signatures.

Aside from researching countermeasures that make the scheme resistant to faults, investigating countermeasures that make the scheme *resilient* to faults could be of equal interest. A fault-resilient countermeasure does not prevent faulty signatures from being collected but from being exploited by hindering at least one step of the universal forgery. While preventing secret extraction or tree grafting would be difficult to achieve without significantly impacting the signing procedure performance (e.g., by replacing the one-time signatures by few-time signatures), a countermeasure that makes path seeking hard to find may reveal to be effective. Such a direction is left as an open problem.

At last, regarding the offensive side of the attack, as the current work is limited to faulting the hash functions, deriving similar attacks by faulting other subroutines of the scheme may lead to equally critical forgeries. Also, tampering with the control flow of a SPHINCS$^+$ software to force one-time signatures to sign unexpected messages would be an interesting direction to consider. Finally, differential fault attacks to recover secret values is yet another breach to explore.

Acknowledgments

I would like to express my sincerest gratitude to Arjen K. Lenstra for all the feedback he provided to me while I was drafting this paper which has significantly improved the quality of the article. I also feel incredibly indebted to Novak Kaluđerović and Thorsten Kleinjung for engaging in the countless discussions on the topic, Léa Micheloud for their help in stochastic processes, and Nicolas Oberli for his assistance with the experimental setup. Finally, I would like to thank the CHES reviewers for their insightful remarks which made the article more comprehensive.

References

[AE17] Jean-Philippe Aumasson and Guillaume Endignoux. Gravity-SPHINCS. Technical report, National Institute of Standards and Technology, 2017. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[ALCZ20] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. FPGA-based SPHINCS$^+$ implementations: Mind the glitch. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*, pages 229–237. IEEE, 2020.

[BBB$^+$12] Alessandro Barenghi, Guido Marco Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. Injection technologies for fault attacks on microprocessors. In Marc Joye and Michael Tunstall, editors, *Fault Analysis in Cryptography*, Information Security and Cryptography, pages 275–293. Springer, 2012.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51. Springer, Heidelberg, May 1997.

[BH17] 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, *SAC 2017*, volume 10719 of *LNCS*, pages 299–322. Springer, Heidelberg, August 2017.

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 368–397. Springer, Heidelberg, April 2015.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2129–2146. ACM Press, November 2019.

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR TCHES*, 2018(3):21–43, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7267.

[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 - 9th International Conference, PQCrypto 2018*, pages 165–184. Springer, Heidelberg, 2018.

[Cor11] Charles J. Corrado. The exact distribution of the maximum, minimum and the range of multinomial/dirichlet and multivariate hypergeometric frequencies. *Stat. Comput.*, 21(3):349–359, 2011.

[Fel67] William Feller. An introduction to probability theory and its applications. *1, 2nd*, 1967.

[FGT92] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discret. Appl. Math.*, 39(3):207–229, 1992.

[FKN+22] Scott Fluhrer, Stefan Kölbl, Ruben Niederhagen, Joost Rijnevald, Peter Schwabe, Bas Westerbaan, and Thom Wiggers. The SPHINCS+ reference code, accompanying the submission to NIST's Post-Quantum Cryptography project. https://github.com/sphincs/sphincsplus.git, 2022.

[GDD+22] Alagic Gorjan, Apon Daniel, Cooper David, Dang Quynh, Dang Thinh, Kelsey John, Lichtinger Jacob, Miller Carl, Moody Dustin, Peralta Rene, Perlner Ray, Robinson Angela, and Smith-Tone Daniel. Status report on the third round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2022.

[GKPM18] Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on SPHINCS. Cryptology ePrint Archive, Report 2018/674, 2018. https://eprint.iacr.org/2018/674.

[GS97] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability.* American Mathematical Soc., 1997.

[HBD+20] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[HRB13] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for XMSS^{MT}. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar R. Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics - CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings*, volume 8128 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2013.

[KAA17] Mehran Mozaffari Kermani, Reza Azarderakhsh, and Anita Aghaie. 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, 2017.

[KDK$^+$14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014.

[MHS$^+$19] Sarah McCarthy, James Howe, Neil Smyth, Séamus Brannigan, and Máire O'Neill. BEARZ attack FALCON: implementation attacks with countermeasures on the FALCON signature scheme. In Mohammad S. Obaidat and Pierangela Samarati, editors, *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*, pages 61–71. SciTePress, 2019.

[NIS16] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, December 2016.

[NS88] Kazuo Nishimura and Masaaki Sibuya. Occupancy with two types of balls. *Annals of the Institute of Statistical Mathematics*, 40:77–91, 1988.

[Onl22] Online Hash Crack. Benchmark Hashcat RTX 3090. https://www.onlinehashcrack.com/tools-benchmark-hashcat-nvidia-rtx-3090.php, 2022. Accessed: 2022-09-01.

[SPW11] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. *Commun. ACM*, 54(2):100–107, 2011.