A Scalable Fault Countermeasure for SLH-DSA

Trade-offs Between Memory, Performance, and Fault Resilience

Melissa Azouaoui[1], Tobias Schneider[2] and Denise Verbakel[3]

[1] NXP Semiconductors, Hamburg, Germany
[2] NXP Semiconductors, Gratkorn, Austria
[3] NXP Semiconductors, Eindhoven, The Netherlands
firstname.lastname@nxp.com

Abstract. We introduce compressed caching, a scalable and parameterizable countermeasure against grafting tree fault attacks on SLH-DSA. Unlike standard caching, which entails fully caching the WOTS$^+$ signatures and public keys, compressed caching achieves significant memory savings while maintaining strong fault detection capabilities. It can be tuned to achieve a trade-off between caching memory size, fault resilience, and performance, making it well-suited for deployment across devices with varying resource and security constraints. We provide a security and performance analysis of compressed caching and show that it can be configured to achieve high fault detection probability and outperform standard caching, mainly in terms of memory but also in terms of performance. Additionally, we explore granular variants of both standard and compressed caching and study on a finer scale the memory-performance trade-off of both standard and compressed caching. Our results demonstrate that compressed caching is especially advantageous for constrained devices, outperforming standard caching when less than approximately 256 kB of caching memory is available.

Keywords: Post-Quantum Cryptography · SLH-DSA · SPHINCS$^+$ · Fault Injection Attacks · Fault Attack Countermeasures

1 Introduction

In response to the growing threat posed by quantum computing to classical digital signatures, the National Institute of Standards and Technology (NIST) has selected several post-quantum digital signature schemes for standardization [NIS]. One of the selected algorithms is SLH-DSA, which is a stateless hash-based digital signature algorithm derived from the SPHINCS$^+$ framework [BHK+19]. Being a hash-based signature scheme, the security of the scheme is fully based on the cryptographic properties of the hash functions used. This implies that it is resilient against many traditional cryptanalytic attacks; however, it has been shown to be critically vulnerable to fault injection attacks [CMP18, GKPM18, Gen23].

Fault attacks exploit unintended behavior in cryptographic implementations caused by physical disturbances such as voltage glitches or laser injections [CML+11].These attacks can compromise the integrity of digital signatures by manipulating internal computations. In SLH-DSA, fault attacks are especially dangerous because the scheme depends on one-time signatures (which should only be used to sign a single and fixed message) and hash trees (which should stay the same during the signing process).

The grafting tree attack [CMP18] on the SPHINCS framework enables existential forgery (the creation of a valid signature for any chosen message M) once both a valid and a faulty signature for the same message are obtained. The attack works by introducing faults during

the computation of hash trees, which can lead to the reuse of one-time signatures. This reuse violates the core security assumption of one-time signatures, allowing an attacker to construct a counterfeit hash tree that appears validly signed. As a result, the attacker can forge signatures for arbitrary messages, compromising the integrity of the entire scheme. Subsequent studies have confirmed the broad applicability and severity of this attack. Amiet et al. [ALCZ20] replicated the attack on custom hardware implementations, while Genêt's work [Gen23] extended the analysis to SPHINCS$^+$ with randomized signing. These studies revealed that even a single random bit flip during signing can significantly degrade the scheme's security, and that faulty signatures often pass verification undetected.

Despite the critical nature of these findings, the development of effective and efficient countermeasures remains a significant challenge. Castelnovi et al. [CMP18] proposed classical redundancy as a generic defense, while Kermani et al. [KAA17] suggested error-detection mechanisms for hash functions that do not fully cover the SLH-DSA signing procedure. Furthermore, Gavrilan et al. [GOW+24] proposed fault-resilient implementations using threshold techniques in hash functions, and Genêt [Gen23] proposed extensive caching of the one-time signatures to mitigate fault attacks. However, these approaches often incur significant performance or memory overheads, and for some their practicality in constrained environments remains unclear.

Contributions. In this paper, we present compressed caching, a flexible and scalable countermeasure designed to protect SLH-DSA implementations against grafting tree fault attacks. Compared to the WOTS$^+$ caching countermeasure analyzed in [Gen23], compressed caching significantly reduces memory requirements. It offers a tunable trade-off between fault resilience, memory usage, and performance making it adaptable to a wide range of devices and deployment scenarios. This flexibility makes compressed caching particularly suitable for real-world applications where resources are constrained and fault resilience is critical.

Outline. This paper is structured as follows. Section 2 provides a high-level overview of the SLH-DSA scheme, the grafting tree attack, and existing countermeasures. Section 3 introduces the proposed compressed caching countermeasure. Section 4 analyzes its fault resilience against single- and multiple-fault attacks and provides guidance on parametrizing the compressed caching countermeasure to achieve a desired memory to fault resistance trade-off. Section 5 provides the memory footprint, and performance improvements (measured in hash calls) compared to the standard caching countermeasure [Gen23]. Section 6 presents a practical validation of the theoretical performance gains, includes fault injection simulations to confirm the predicted fault detection probabilities and assesses the impact of additional mitigations against multiple-fault adversaries. Finally, Section 7 briefly explores granular variants of both the standard and compressed caching countermeasures, with the main goal of clarifying when compressed caching is most beneficial.

2 Background

In the following sections and throughout the remainder of this paper, we adopt the same or similar notational conventions as those presented in [Gen23] in order to maintain consistency with existing literature. Nevertheless, as prior works predominantly focus on SPHINCS$^+$, we adapt the notation to conform to the SLH-DSA standard [NIS24].

2.1 SLH-DSA

SLH-DSA is a digital signature scheme that ensures data integrity, authenticates the identity of the signer, and supports non-repudiation. It is derived from SPHINCS$^+$, which was

selected for standardization under the NIST Post-Quantum Cryptography Standardization initiative. SLH-DSA is a stateless hash-based signature scheme, distinguishing itself from related schemes such as LMS [MCF19] and XMSS [HBG+18] by eliminating the need for state management. This stateless design is achieved through the integration of a broad few-time signature (FTS) scheme, layered with a hypertree structure composed of one-time signature (OTS) schemes. Specifically, the FTS component is implemented using FORS, while the hypertree is constructed using XMSS trees utilizing WOTS$^+$ signatures. Below, we provide a brief overview of the three main components of SLH-DSA. For comprehensive technical details or more information on these building blocks, we refer the reader to the SLH-DSA standard [NIS24].

Definitions and notations. SLH-DSA uses a variety of hash functions and pseudo-random functions across its APIs to support several cryptographic operations. These include generating the randomizer R for randomized message hashing, generating a digest of the message to be signed, generating secret values for WOTS$^+$ and FORS, and compressing input data of ℓn, $2n$, or n bytes into n-byte outputs. An overview of the hash and pseudo-random functions used for each of these purposes is provided in Table 1, where $\mathbb{B} = \{0, \ldots, 255\}$ denotes the set of all bytes, $\mathbb{B}^n$ denotes the set of byte strings of length n bytes and $\mathbb{B}^*$ denotes the set of all byte strings.

Table 1: Hash functions and pseudo-random functions in SLH-DSA.

Function	**Input size**	**Output size**
$\mathbf{PRF}_{msg}(\mathbf{SK.prf}, opt_{rand}, M)$	$\mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^*$	$\mathbb{B}^n$
$\mathbf{H}_{msg}(R, \mathbf{PK.seed}, \mathbf{PK.root}, M)$	$\mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^*$	$\mathbb{B}^m$
$\mathbf{PRF}(\mathbf{PK.seed}, \mathbf{SK.seed}, \mathbf{ADRS})$	$\mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^{32}$	$\mathbb{B}^n$
$\mathbf{T}_\ell(\mathbf{PK.seed}, \mathbf{ADRS}, M_\ell)$	$\mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{\ell n}$	$\mathbb{B}^n$
$\mathbf{H}(\mathbf{PK.seed}, \mathbf{ADRS}, M_2)$	$\mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{2n}$	$\mathbb{B}^n$
$\mathbf{F}(\mathbf{PK.seed}, \mathbf{ADRS}, M_1)$	$\mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^n$	$\mathbb{B}^n$

The SLH-DSA public key is constructed using two n-byte elements: **PK.seed** || **PK.root**. The secret key contains $4n$ bytes: **SK.seed** || **SK.prf** || **PK.seed** || **PK.root**. Furthermore, M stands for the provided message block(s) and opt_{rand} for n bytes of randomness (or **PK.seed** for the deterministic SLH-DSA variant). The SLH-DSA **ADRS** structure is 32 bytes for SHAKE variants and 22 bytes for SHA2 variants. It encodes several elements such as the leaf index (which identifies individual leaves within a Merkle tree [Mer79]), the tree index (which distinguishes among the multiple Merkle trees in the hypertree), and domain separation fields for distinguishing functional contexts.

Forest of Random Subsets (FORS). FORS is the few-time signature scheme used to sign message digests in SLH-DSA. Unlike WOTS$^+$, where key reuse can lead to feasible forgeries, the security of a FORS key degrades gradually with repeated use. FORS operates with two parameters:

- k: the number of FORS trees;
- $t = 2^a$: the number of leaves in a FORS tree of height a.

The private key consists of k sets of t n-byte strings, all pseudo-randomly derived from the secret seed **SK.seed**. Each set is organized into a Merkle tree, and the roots of these trees are hashed together to form the FORS public key. A signature on a $k \cdot a$-bit message digest includes k elements from the private keys - one from each set, determined by a bits of the digest - along with their corresponding authentication paths. The *authentication path* consists of the sibling nodes along the path from a selected leaf to the root of the

Merkle tree. These nodes enable recomputation of the root, allowing verification of the signature without iterating through the entire tree.

Winternitz One-Time Signature Plus (WOTS$^+$). WOTS$^+$ is the one-time signature scheme used within SLH-DSA to authenticate subtrees by signing their respective root nodes. It utilizes the chain function, which repeatedly applies a hash function to an input, using each output as the input for the next iteration, until a predefined number of hashing operations has been completed. The WOTS$^+$ scheme requires three parameters:

- n: the security parameter, length in bytes of the WOTS$^+$ elements/messages;
- $lg_w = \log_2(w)$: the number of bits that are encoded by each hash chain of length w;
- $\ell = \ell_1 + \ell_2$: number of secret and public values in WOTS$^+$.

A WOTS$^+$ private key comprises ℓ secret values of length n, derived from the secret seed **SK.seed** using a pseudo-random function. Each secret value is used to generate a chain of length w hashes via a chaining function, with the final values of these chains forming the public values. The WOTS$^+$ public key is then computed as the hash of these public values. To sign a message, it is first converted into an array of ℓ_1 base-w integers, followed by a checksum of ℓ_2 base-w integers. The signature consists of selected intermediate values from the chains, corresponding to the message and checksum integers.

eXtended Merkle Signature Scheme (XMSS). XMSS extends the WOTS$^+$ signature scheme to enable the signing of multiple messages. It introduces an additional parameter, h', which specifies the height of a Merkle tree. This tree structure allows to authenticate $2^{h'}$ WOTS$^+$ public keys using a single n-byte XMSS public key, which corresponds to the root of the Merkle tree. An XMSS signature comprises a WOTS$^+$ signature and an authentication path, with a total size of $(\ell + h') \cdot n$ bytes.

SLH-DSA hypertree structure. Because signing a large number of FORS public keys needs a large number of WOTS$^+$ keys, using just one XMSS key pair is not practical within the limits set by the standard. To address this, SLH-DSA employs a hypertree structure, which consists of multiple layers of Merkle trees. In this configuration, XMSS keys at the lowest layer are designated for signing FORS public keys, while XMSS keys at higher layers are used to authenticate the XMSS public keys of the layer directly below them. The hypertree structure needs two additional parameters:

- d: number of layers of XMSS trees;
- $h = d \cdot h'$: the total height of the hypertree (d layers of XMSS trees of height h').

SLH-DSA parameter sets. The parameter sets chosen by NIST for the SLH-DSA standard are listed in Table 2. These parameter sets were selected to ensure existential unforgeability under chosen message attacks (EUF-CMA), assuming that each key pair is used to sign no more than 2^{64} messages.

SLH-DSA sign. Algorithm 1 shows a simplified version of the SLH-DSA signing process. For the full detailed algorithm, we refer to Algorithm 19 of the SLH-DSA standard [NIS24]. Note that for the top layer, so on layer $d-1$, `xmss_pkFromSig` is not invoked. From this point forward, we refer to the following functions: `fors_sign` (creation of a FORS signature), `fors_pkFromSig` (creation of a FORS public key), `xmss_sign` (creation of an XMSS signature, which is a WOTS$^+$ signature and its associated authentication path), `xmss_pkFromSig` (creation of an XMSS public key), and `slh_sign` (the SLH-DSA signing algorithm).

Table 2: SLH-DSA parameter sets standardized by NIST.

Parameter set	n	ℓ	h	d	h'	a	t	k	lg_w	w
SLH-DSA-SHA2/SHAKE-128s	16	35	63	7	9	12	2^{12}	14	4	16
SLH-DSA-SHA2/SHAKE-128f	16	35	66	22	3	6	2^6	33	4	16
SLH-DSA-SHA2/SHAKE-192s	24	51	63	7	9	14	2^{14}	17	4	16
SLH-DSA-SHA2/SHAKE-192f	24	51	66	22	3	8	2^8	33	4	16
SLH-DSA-SHA2/SHAKE-256s	32	67	64	8	8	14	2^{14}	22	4	16
SLH-DSA-SHA2/SHAKE-256f	32	67	68	17	4	9	2^9	35	4	16

Algorithm 1 `slh_sign`$(M, \mathbf{SK}, addrnd)$

$\mathbf{SIG}_R \leftarrow \mathbf{PRF}_{msg}(\mathbf{SK.prf}, addrnd, M)$
$md, idx_{tree}, idx_{leaf}, \mathbf{ADRS} \leftarrow \mathbf{H}_{msg}(\mathbf{SIG}_R, \mathbf{PK.seed}, \mathbf{PK.root}, M)$
$\mathbf{SIG}_{FORS} \leftarrow$ `fors_sign`$(md, \mathbf{SK.seed}, \mathbf{PK.seed}, \mathbf{ADRS})$
$\mathbf{PK}_{FORS} \leftarrow$ `fors_pkFromSig`$(\mathbf{SIG}_{FORS}, md, \mathbf{PK.seed}, \mathbf{ADRS})$
$\mathbf{SIG}_{XMSS,0} \leftarrow$ `xmss_sign`$(\mathbf{PK}_{FORS}, \mathbf{SK.seed}, idx_{leaf}, \mathbf{PK.seed}, \mathbf{ADRS})$
$root \leftarrow$ `xmss_pkFromSig`$(idx_{leaf}, \mathbf{SIG}_{XMSS}, \mathbf{PK}_{FORS}, \mathbf{PK.seed}, \mathbf{ADRS})$
for j **from** 1 **to** $d-1$ **do**
    Update idx_{leaf}, idx_{tree} and **ADRS** to fit the current layer
    $\mathbf{SIG}_{XMSS,j} \leftarrow$ `xmss_sign`$(root, \mathbf{SK.seed}, idx_{leaf}, \mathbf{PK.seed}, \mathbf{ADRS})$
    if $j < d-1$ **then**
        $root \leftarrow$ `xmss_pkFromSig`$(idx_{leaf}, \mathbf{SIG}_{XMSS,j}, root, \mathbf{PK.seed}, \mathbf{ADRS})$
$\mathbf{SIG} \leftarrow \mathbf{SIG}_R \,||\, \mathbf{SIG}_{FORS} \,||\, \mathbf{SIG}_{XMSS,0} \,||\, \cdots \,||\, \mathbf{SIG}_{XMSS,d-1}$

SLH-DSA hash function calls. Since SLH-DSA is a hash-based signature scheme, its core and main operation is hashing. Below, we outline the number of hash function calls required across various sub-functions of the SLH-DSA signing routine. We define the total hash calls in `fors_sign` $\#\text{Total}^{FS}$ and in `fors_pkFromSig` $\#\text{Total}^{Fpk}$ as:

$$\#\text{Total}^{FS} = k\,(t-1)\cdot\mathbf{F} + k\,(t-a-1)\cdot\mathbf{H} + kt\cdot\mathbf{PRF}$$
$$\#\text{Total}^{Fpk} = k\cdot\mathbf{F} + ka\cdot\mathbf{H} + \mathbf{T}_k$$

Likewise, the average number of hash calls in `xmss_sign` are incorporated in $\#\text{Total}^{XS}$ and the ones for `xmss_pkFromSig` in $\#\text{Total}^{Xpk}$. These are defined as follows:

$$\#\text{Total}^{XS} = \left(\left(2^{h'} - \frac{1}{2}\right)\ell\,(w-1)\right)\cdot\mathbf{F} + \left(2^{h'} - h' - 1\right)\cdot\mathbf{H} + \left(2^{h'} - 1\right)\cdot\mathbf{T}_\ell + 2^{h'}\ell\cdot\mathbf{PRF}$$
$$\#\text{Total}^{Xpk} = \frac{\ell\,(w-1)}{2}\cdot\mathbf{F} + h'\cdot\mathbf{H} + \mathbf{T}_\ell$$

Using these equations, the average total hash calls done within SLH-DSA signing $\#\text{Total}^{S}$, may be expressed as:

$$\#\text{Total}^{S} = \mathbf{PRF}_{msg} + \mathbf{H}_{msg} + \#\text{Total}^{FS} + \#\text{Total}^{Fpk} + d\cdot\#\text{Total}^{XS} + (d-1)\cdot\#\text{Total}^{Xpk}$$

2.2 Grafting tree attack on SLH-DSA

The grafting tree attack on SPHINCS$^+$ was first proposed by Castelnovi et al. in [CMP18]. Its basic idea is to force one of the one-time signature schemes inside the tree to sign two (or multiple) different messages via fault injection. Given these signatures, it is then possible to extract information about the WOTS$^+$ secret key, which is used to authenticate

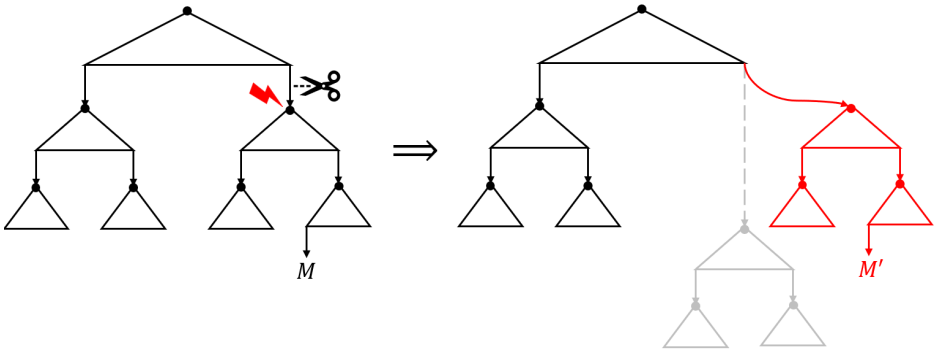


Figure 1: Illustration of the grafting tree fault attack on SLH-DSA. Inspired by Figure 3 in [GKPM18].

an attacker-generated sub-tree and forge signatures. The high-level idea of such a grafting tree attack is illustrated in Figure 1.

One of the main strengths of the attack is its very relaxed fault model. In the most basic case, it requires a single, random value fault injected during signature generation on a wide attack surface. In particular, Castelnovi et al. note that faulting the authentication path will lead to a signature useful for an attacker, which is still valid and as such could not easily be detected via signature verification. While the initial paper [CMP18] focused on the deterministic variant of SPHINCS$^+$ (relying on signing the same message multiple times to ensure the same path through the tree) Genêt generalizes the grafting tree attack in [Gen23] to randomized signing.

Overall, the grafting tree attack represents one of the most powerful fault attacks on SPHINCS$^+$ and SLH-DSA due to its relaxed attacker model and ability to avoid detection via signature verification. Therefore, any practical implementation of SLH-DSA for which fault attacks are in scope is required to integrate dedicated countermeasures to thwart it.

2.3 Grafting tree attack countermeasures for SLH-DSA

The grafting tree attack on SLH-DSA is a powerful attack with a large attack surface. An ad-hoc countermeasure against it would imply recomputing the SLH-DSA signature generation and comparing the output signatures. A second countermeasure proposed in the literature involves caching the WOTS$^+$ signatures. Below, we provide a brief overview of recomputation and caching, accompanied by an analysis of their respective hash function call counts and memory usage for caching.

Recomputation. Recomputation is an ad-hoc fault attack countermeasure. It would involve computing an SLH-DSA signature twice for the same inputs (message and randomness) and verifying that the output signatures are identical. This protects against a single fault injected during a signature generation and can be generalized to protect against x faults by computing the signature $x + 1$ times and ensuring that all output signatures are equal. Such a recomputation can also be performed more granularly by recomputing each layer or after each WOTS$^+$ or XMSS signature. The average total hash function calls for `slh_sign` protected with double computation is simply:

$$\#\text{Total}^S_{recomp} = 2 \cdot \#\text{Total}^S$$

Caching. The caching countermeasure as described and analyzed by Genêt [Gen23] consists of caching the WOTS$^+$ signatures along with their corresponding WOTS$^+$ public

keys for a number of layers starting from the top of the hypertree. Since the cached WOTS$^+$ signatures are no longer computed and simply fetched from the cache and appended to the signature, the corresponding WOTS$^+$ key pairs are no longer vulnerable to the grafting tree attack. This also naturally accelerates the signing process. Furthermore, the computation of the authentication path for the XMSS signature associated with a cached WOTS$^+$ signature requires only $\left(2^{h'}-1\right)$ calls to $\mathbf{H}$ since all WOTS$^+$ public keys for that tree are cached. However, the downside of this method is the impractical amount of memory required for SLH-DSA parameter sets.

Caching is configured by layer, i.e., the WOTS$^+$ signatures and public keys are cached for the top c layers. Similarly to [Gen23], we denote this number of hash function calls required to compute the authentication path from the cached WOTS$^+$ public keys as:

$$\#\text{Total}^{\widetilde{XS}} = \left(2^{h'}-1\right) \cdot \mathbf{H}$$

Note that $\widetilde{XS}$, as opposed to XS, indicates a cached layer. We assume that non-cached layers and any other hash calls not involved in a cached layer are recomputed to protect against faults[1]. The average number of hash function calls in `slh_sign` protected with caching the top c layers (with $c \in \{0, 1, 2, \ldots, d\}$ and $c = 0$ meaning no caching) and recomputing the other layers is:

$$\#\text{Total}^{S}_{caching}(c) = \begin{cases} \#\text{Total}^{S}_{recomp} & \text{if } c = 0 \\ 2 \cdot \Big(\mathbf{PRF}_{msg} + \mathbf{H}_{msg} + \#\text{Total}^{FS} + \#\text{Total}^{Fpk} + (d-c) \cdot \#\text{Total}^{XS} & \\ \quad + (d-c) \cdot \#\text{Total}^{Xpk}\Big) + c \cdot \#\text{Total}^{\widetilde{XS}} & \text{if } c \geq 1 \end{cases}$$

The blue part of the hash count denotes the operations of the signature generation that are computed for the cached layers and hence not recomputed. The memory required in bytes to cache the WOTS$^+$ signatures and their public keys for the c top layers is equal to:

$$\#\text{Mem}^{S}_{caching}(c) = \frac{2^{h'}\left(2^{ch'}-1\right)}{2^{h'}-1}(\ell+1)\,n \text{ bytes} \tag{1}$$

Table 3 presents the memory requirements for all NIST-defined SLH-DSA parameter sets, evaluated across varying numbers of cached layers c. As noted in [Gen23], the cost in terms of memory rapidly becomes prohibitive and even caching the top few layers can be impractical for some devices[2]. For instance, while caching the first layer requires ≈ 4.6 kB for the 128f parameter set, it requires close to 300 kB for the 128s parameter set. Caching only two layers requires over 40 kB for all parameter sets and up to 328 MB for the 192s parameter set. In the remainder of this paper we refer to this countermeasure as standard caching.

3 Compressed Caching Description

As is clear from Section 2.3, the memory required to cache the WOTS$^+$ signatures and public keys for many layers of an SLH-DSA hypertree is practically infeasible for a wide

[1]While the work in [Gen23] does not focus on recomputation, we consider recomputation as a reasonable ad-hoc countermeasure to protect non-cached layers. We also assume that $\mathbf{PRF}_{msg}$, $\mathbf{H}_{msg}$ and the FORS signature are recomputed to hinder other attacks that may target the message or the FORS signature.

[2]Note that some numbers in our Table 3 differ from the ones in Table 10 in [Gen23], simply due to the fact that we use SLH-DSA parameters sets as defined by NIST, whereas most previous publications deal with SPHINCS$^+$ parameters. Table 10 in [Gen23] also did not seem to include the WOTS$^+$ public keys.

Table 3: Memory required for the SLH-DSA layered caching countermeasure.

Parameter set	Memory (bytes)					
	$c=1$	$c=2$	$c=3$	$c=4$	...	$c=d$
SHA2/SHAKE-128s	2.95×10^5	1.51×10^8	7.75×10^{10}	3.97×10^{13}	...	5.32×10^{21}
SHA2/SHAKE-128f	4.61×10^3	4.15×10^4	3.36×10^5	2.70×10^6	...	4.86×10^{22}
SHA2/SHAKE-192s	6.39×10^5	3.28×10^8	1.68×10^{11}	8.59×10^{13}	...	1.15×10^{22}
SHA2/SHAKE-192f	9.98×10^3	8.99×10^4	7.29×10^5	5.84×10^6	...	1.05×10^{23}
SHA2/SHAKE-256s	5.57×10^5	1.43×10^8	3.67×10^{10}	9.38×10^{12}	...	4.03×10^{22}
SHA2/SHAKE-256f	3.48×10^4	5.92×10^5	9.50×10^6	1.52×10^8	...	6.85×10^{23}

range of devices. Caching even the top few layers can be prohibitive depending on the parameter set. In this section we describe a more practical caching countermeasure, that we intuitively call compressed caching, which offers a trade-off between memory requirements, speed and fault resilience.

The proposed so-called compressed caching is conceptually simple. Instead of caching the full WOTS$^+$ signatures and corresponding public keys, we propose to cache the WOTS$^+$ signatures in a compressed form: a truncated hash of the WOTS$^+$ signature. We denote this truncated hash as $\mathsf{Trunc}_g(\mathcal{H}(X))$. $\mathsf{Trunc}_g(X) : \mathbb{B}^* \rightarrow \mathbb{B}^g$ is defined as a function that outputs the leftmost g bytes of the input byte string X [NIS24]. $\mathcal{H}$ denotes the hash function selected to implement the compressed caching, e.g., the one used throughout SLH-DSA and defined by the parameter set. Note that the input X should include at a minimum the full WOTS$^+$ signature, since the goal of the attacker is to inject a fault that leads to the WOTS$^+$ secret key signing an input it was not meant to sign and hence produce a different WOTS$^+$ signature. In the remainder of this paper, unless otherwise stated, X includes only the WOTS$^+$ signature. Compressed caching allows to perform an integrity check on the generated WOTS$^+$ signatures by comparing their truncated hashes during runtime to the corresponding values in the cache.

We first provide a description of how the cache is generated. Note that the generation of this cache can be done at the same time as key generation.We then provide a description of the signing procedure using the compressed caching countermeasure to protect the top c' layers and using recomputation for the remaining bottom layers.

3.1 Cache generation process

Let c' be the number of layers for which the compressed caching is used. During key generation, instead of storing each WOTS$^+$ signature and public key for all target layers as done in the standard caching countermeasure, compressed caching stores a truncated hash with an output of g bytes of each WOTS$^+$ signature for the top c' layers. This is shown in the pseudo-code in Algorithm 2.

Algorithm 2 `compressed_cache`$(\mathbf{SK}, c', g)$

for j **from** $d-1$ **down to** $d-c'$ **do**
  $base_idx \leftarrow \dfrac{2^{h' \cdot (d-j)} - 2^{h'}}{2^{h'} - 1}$
  for T **from** 0 **to** $2^{h' \cdot (d-j-1)} - 1$ **do**
    for i **from** 0 **to** $2^{h'} - 1$ **do**
      $root \leftarrow \texttt{xmss_pk}(\mathbf{SK}, j-1, 2^{h'} \cdot T + i)$
      $wots_sig \leftarrow \texttt{wots_sign}(\mathbf{SK}, j, T, i, root)$
      $idx \leftarrow base_idx + T \cdot 2^{h'} + i$
      $cache[idx] \leftarrow \mathsf{Trunc}_g(\mathcal{H}(wots_sig))$

Caching is done for each layer j from the top of the hypertree until layer $d - c'$. For leaf i of each tree T of each layer j, its corresponding root indexed $2^{h'} \cdot T + i$ from the layer below $(j-1)$ is computed at line 5. We denote by `xmss_pk` the function that computes the root of an XMSS tree in the SLH-DSA hypertree given the secret key **SK**, the layer index and tree index. This XMSS root is then signed by leaf i at line 6. We denote by `wots_sign` the function that produces the WOTS$^+$ signature given the secret key **SK**, layer, tree, and leaf indexes and the input to be signed. To access the correct elements of the cache, the cached WOTS$^+$ signatures are indexed from top to bottom. The cache comprises $\frac{2^{h'}\left(2^{c' \cdot h'} - 1\right)}{2^{h'} - 1}$ elements and the indexes are computed with the help of lines 2 and 7. Finally, the truncated hash of the WOTS$^+$ signature is computed and stored at the corresponding index in the cache at line 8.

3.2 Protected signature generation using compressed caching

Pseudo-code for the protected signing is given in Algorithm 3. During signing, fault protection is applied differently across the layers. We assume that all bottom layers from 0 up to $d - c'$ are protected using ad-hoc standard recomputation: addresses/indexes, (FORS, XMSS) signatures, and tree roots are recomputed and compared to ensure correctness for subsequent operations. For the top c' layers, compressed caching is applied, as illustrated in Algorithm 3, lines 13 to 17. Instead of recomputation, the truncated hash of the WOTS$^+$ signature of the current layer is compared to its cached version.

Algorithm 3 `slh_sign_cc`$(M, \mathbf{SK}, addrnd, cache, c')$

1: $\mathbf{SIG}_R \leftarrow \mathbf{PRF}_{msg}(\mathbf{SK.prf}, addrnd, M)$
2: $md, idx_{tree}, idx_{leaf}, \mathbf{ADRS} \leftarrow \mathbf{H}_{msg}(\mathbf{SIG}_R, \mathbf{PK.seed}, \mathbf{PK.root}, M)$
3: $\mathbf{SIG}_{FORS} \leftarrow$ `fors_sign`$(md, \mathbf{SK.seed}, \mathbf{PK.seed}, \mathbf{ADRS})$
4: $\mathbf{PK}_{FORS} \leftarrow$ `fors_pkFromSig`$(\mathbf{SIG}_{FORS}, md, \mathbf{PK.seed}, \mathbf{ADRS})$
5: $\mathbf{SIG}_{XMSS,0} \leftarrow$ `xmss_sign`$(\mathbf{PK}_{FORS}, \mathbf{SK.seed}, idx_{leaf}, \mathbf{PK.seed}, \mathbf{ADRS})$
6: $root \leftarrow$ `xmss_pkFromSig`$(idx_{leaf}, \mathbf{SIG}_{XMSS}, \mathbf{PK}_{FORS}, \mathbf{PK.seed}, \mathbf{ADRS})$
7: Recompute lines 1 to 6 and compare results to detect faults
8: **for** j **from** 1 **to** $d-1$ **do**
9: Update idx_{leaf}, idx_{tree} and **ADRS** to fit the current layer
10: $\mathbf{SIG}_{XMSS,j} \leftarrow$ `xmss_sign`$(root, \mathbf{SK.seed}, idx_{leaf}, \mathbf{PK.seed}, \mathbf{ADRS})$
11: **if** $j < d-1$ **then**
12: $root \leftarrow$ `xmss_pkFromSig`$(idx_{leaf}, \mathbf{SIG}_{XMSS,j}, root, \mathbf{PK.seed}, \mathbf{ADRS})$
13: **if** $j \geq d - c'$ **then**
14: $base_idx \leftarrow \dfrac{2^{h' \cdot (d-j)} - 2^{h'}}{2^{h'} - 1}$
15: $idx \leftarrow base_idx + idx_{tree} \cdot 2^{h'} + idx_{leaf}$
16: $tmp \leftarrow \mathsf{Trunc}_g(\mathcal{H}(\mathbf{SIG}_{\mathrm{WOTS}^+,j}))$
17: Compare tmp and $cache[idx]$ to detect faults
18: **else**
19: Recompute lines 9 to 12 and compare results to detect faults
20: $\mathbf{SIG} \leftarrow \mathbf{SIG}_R \,||\, \mathbf{SIG}_{FORS} \,||\, \mathbf{SIG}_{XMSS,0} \,||\, \cdots \,||\, \mathbf{SIG}_{XMSS,d-1}$

We choose to store the truncated hash of the WOTS$^+$ signature. Alternatively, an implementer may opt to use the full XMSS signature, which consists of a WOTS$^+$ signature and an authentication path. Faults on the authentication path affect the root computed in the following layer, leading to an incorrect root and, consequently, an incorrect WOTS$^+$ signature which does not correspond to the cached truncated hash up to a probability depending on g.

The standard caching as explained in Section 2.3 also caches the WOTS$^+$ public keys for the top c layers in order to speed up signing for the cached layers, namely the derivation of the authentication path. The compressed caching only stores truncated hashes of the WOTS$^+$ signatures, and therefore, the cached data does not enable a more efficient derivation of the authentication path. Hence, it might seem counter-intuitive or unexpected that compressed caching can use less memory *and* provide better performance than standard caching. However, we will show in the next sections that due to the significant reduction in memory requirements, we are able to cache more layers and depending on the SLH-DSA parameter sets and parameters of the countermeasure, compressed caching can be beneficial. Before discussing the memory improvements and performance impact of compressed caching compared to standard caching, we first analyze its security in the next section.

4 Security Analysis of Compressed Caching

In this section, we first examine the compressed caching countermeasure's resilience against a single fault during each execution of the signature generation routine. We then broaden the analysis to consider multiple faults per execution and explore attack vectors that the compressed caching countermeasure may have introduced.

4.1 Notations

To enhance the clarity of the analysis presented in this section, we first introduce a simplified visual representation in Equation 2 of the compressed caching countermeasure (its process during signature generation). Then, we introduce a notation to distinguish between different kinds of attacks and attacker capabilities.

$$\tilde{X} = \texttt{wots_sign}(\mathbf{SK}, root) \rightarrow \mathsf{Trunc}_g(\mathcal{H}(\tilde{X})) \rightarrow \boxed{=?} \leftarrow \{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{cache} \quad (2)$$

In Equation 2, X denotes a WOTS$^+$ signature[3] which is produded by WOTS$^+$ signing the root of a tree in a layer below using a WOTS$^+$ secret key derived from the SLH-DSA secret seed $\mathbf{SK}$ (we make abstraction of the indices for conciseness). For compressed caching, X is hashed and its truncated hash is stored in the cache. $\{x\}_{cache}$ denotes that x is a static value stored in the cache. $\tilde{X}$ denotes the corresponding WOTS$^+$ signature computed during signature generation. The $\sim$ denotes the fact that X could be incorrect due to a fault attack, e.g., if a faulty *root* is signed or if the signing process in `wots_sign` is corrupted[4], e.g., by skipping a hash. During signing, for the layers protected with compressed caching, each generated WOTS$^+$ signature $\tilde{X}$ is hashed and its truncated hash $\mathsf{Trunc}_g(\mathcal{H}(\tilde{X}))$ is compared with its expected value in the cache $\{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{cache}$. A fault is detected if the two values are not identical (line 17 in Algorithm 3).

To label attacks we use the following notation $\mathcal{A}^{m,F}_{f_0,f_1,\ldots,f_{m-1}}$, where m is the number of faults the attacker injects during a single execution of the signing procedure, $f_i, \forall 0 \leq i < m$ will encode the type of fault injected and F is the total number of executions faulted. For instance, $\mathcal{A}^{1,20}_{f_0}$ refers to an attacker injecting one fault for 20 executions of the signature generation. Accordingly, the total number of faults injected is $m \cdot F$. In this analysis, we consider the following types of relevant faults: random faults (r), skipping faults (s) and fully controlled faults (c). A random fault entails faulting an intermediate to a random value and a skipping fault entails skipping an instruction, operation or function call chosen by the attacker. The latter, controlled faults, entail that the attacker is able to choose the

[3] As mentioned earlier in the paper, X could include other intermediate values during signing but at a minimum the WOTS$^+$ signature. In this description, for simplicity and clarity we take X as being the WOTS$^+$ signature.

[4] Recall that a direct fault on X (the WOTS$^+$ signature) is not useful to an attacker.

exact effect of the injected fault on the faulted intermediate value or instruction. We use $\Pr[\mathcal{A}^{m,F}_{f_0,f_1,\ldots,f_{m-1}}]$ to denote the success probability of attacker $\mathcal{A}^{m,F}_{f_0,f_1,\ldots,f_{m-1}}$.

4.2 Resistance to single-fault attacks

The fault resilience of the compressed caching depends on the value of the parameter g, which dictates the output size in bytes of the truncated hash. Namely, if g is too small, the collision probability over the truncated hash is relatively high, which in turn makes bypassing the countermeasure easier. In the following, we analyze more concretely the security of the compressed caching countermeasure against the grafting tree attack assuming a single fault per signing and under different fault models.

The grafting tree attack as described in [Gen23] involves many steps. Some of which are essentially brute-force searches. Such steps are not impacted by the fault injection difficulty or caching. Instead what caching or other fault countermeasures mainly alter or lower is the probability for an attacker to receive as output an exploitable faulty WOTS$^+$ signature after a fault injection. For example, this probability is 0 for the WOTS$^+$ signatures cached using standard caching as described in [Gen23], since the cached WOTS$^+$ signatures are no longer being computed and simply fetched from the cache and appended to the signature.

The attacker only requires a single faulty signature to achieve forgery with a non-negligible probability. This forgery probability can be improved with more faulty WOTS$^+$ signatures for the same WOTS$^+$ key pair. For the following analysis, to get a lower bound on the effectiveness of the compressed caching countermeasure, we assume the worst case, i.e., that one single faulty WOTS$^+$ signature compromises the whole scheme. In addition to assuming perfect fault injection probability, i.e., that without any countermeasure an attacker will always receive an exploitable faulty WOTS$^+$ signature, we first analyze random and skipping faults (which are more realistic) and then analyze controlled faults (which assume a very strong attacker model).

Random or skipping fault - $\mathcal{A}^{1,F}_r$ or $\mathcal{A}^{1,F}_s$. An attacker injects random faults or skipping faults during an SLH-DSA signature generation, with the goal of inducing a faulty WOTS$^+$ signature at a specific layer known to the attacker. This is shown in Equation 3. Note that, using one fault to skip the comparison does not lead to an exploitable output, since at least one fault is required to induce a faulty WOTS$^+$ signature. However, the attacker cannot control or predict the effect of the fault on the resulting WOTS$^+$ signature before the fault is injected and the output is observed, since the intermediates depend on the WOTS$^+$ secret key.

$$\tilde{X} = \overset{\lightning\,\mathcal{A}^1_s}{\texttt{wots_sign}}\,(\mathbf{SK}, \overset{\lightning\,\mathcal{A}^1_r}{root}) \rightarrow \mathsf{Trunc}_g(\mathcal{H}(\tilde{X})) \rightarrow \boxed{=?} \leftarrow \{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{cache} \quad (3)$$

Since the compressed caching is based on $\mathsf{Trunc}_g(\mathcal{H}(X))$ providing an output of g bytes, the probability that an attacker succeeds in bypassing it after F fault injections is:

$$\Pr[\mathcal{A}^{1,F}_r] = \Pr[\mathcal{A}^{1,F}_s] = 1 - \left(1 - \frac{1}{2^{8g}}\right)^F$$

This corresponds to the probability that a collision of the form $\mathsf{Trunc}_g(\mathcal{H}(X)) = \mathsf{Trunc}_g(\mathcal{H}(X'))$ for $X \neq X'$ occurs, where X includes the correct WOTS$^+$ signature and X' includes the faulty WOTS$^+$ signature, after F fault injections.

Controlled fault - $\mathcal{A}^{1,F}_c$. In the following, we assume an attacker able to inject a fault during an SLH-DSA signature generation at the exact location they want to target and with full control over the exact fault they induce. We will discuss in the following how

an attacker with fully controlled faults does not necessarily have a significant advantage compared to an attacker able to inject random or skipping faults.

In the following we denote the ℓ elements of the WOTS$^+$ signature X as $X = [X_0, X_1, \ldots, X_{\ell-1}]$. Each X_i is defined as $X_i = \mathcal{H}^{digest_i}(sk_i)$, where $\mathcal{H}^{digest_i}$ denotes $\mathcal{H}$ applied iteratively $digest_i$ times and sk_i denotes one element of the target WOTS$^+$ secret key. Recall that to bypass the compressed caching countermesaure, an attacker would aim to inject a fault to cause a collision of the form $\mathsf{Trunc}_g(\mathcal{H}(X)) = \mathsf{Trunc}_g(\mathcal{H}(X'))$ with $X \neq X'$. A useful fault would translate to a lower $digest_i$ value[5]. An attacker with fully controlled faults, could aim to find any fault effect and location during the execution of the signing routine such that the digest element is corrupted to $digest'_i = digest_i - \delta$ with $\delta > 0$. To bypass the compressed caching countermeasure, the attacker would need to find any fault effect and location corresponding to $\delta > 0$ or alternatively find $\delta > 0$ (and skip δ hash calls) such that:

$$\begin{aligned}&\mathsf{Trunc}_g(\mathcal{H}(X_0, ..., X_i, ..., X_{\ell-1})) = \mathsf{Trunc}_g(\mathcal{H}(X_0, ..., X'_i, ..., X_{\ell-1}))\\ \Leftrightarrow &\mathsf{Trunc}_g(\mathcal{H}(X_0, ..., \mathcal{H}^{digest_i}(sk_i), ..., X_{\ell-1})) = \mathsf{Trunc}_g(\mathcal{H}(X_0, ..., \mathcal{H}^{digest_i-\delta}(sk_i), ..., X_{\ell-1}))\end{aligned}$$

which, without knowledge of the WOTS$^+$ secret key sk, amounts to a second-preimage search on the truncated hash output. While finding an X'_i that verifies the above equation is possible in an offline setting (i.e., without access to the device holding the WOTS$^+$ secret key sk), up to a probability/effort depending on g, finding the corresponding δ requires online access to the device. This extends to the general case of a corrupted digest (for e.g., when the root to be signed has been faulted), since determining whether the corrupted digest leads to a truncated hash collision requires evaluating the WOTS$^+$-secret-key-dependent hashes. Accordingly, predicting the effect of a fully controlled fault on the resulting WOTS$^+$ signature cannot be done offline and the probability of success in bypassing the compressed caching countermeasure is the same as in the case of skipping or random faults. As a result, we have that:

$$\Pr[\mathcal{A}_r^{1,F}] = \Pr[\mathcal{A}_s^{1,F}] = \Pr[\mathcal{A}_c^{1,F}] = \Pr[\mathcal{A}_*^{1,F}] = 1 - \left(1 - \frac{1}{2^{8g}}\right)^F \tag{4}$$

where a $*$ denotes any fault in the set of faults we consider, i.e., $* \in \{r, s, c\}$.

Selection of g and combination with a fault counter. In the following, let F_{max} denote the maximum number of detected faults before the device aborts all signing, refreshes the key, reduces throughput or activates additional countermeasures. The probability p_{max} is the maximum target success probability of an attacker $\mathcal{A}_*^{1,F_{max}}$.

Each time a fault is injected inducing an incorrect WOTS$^+$ signature, the truncated hashes are not equal and the fault is detected with probability $\left(1 - \frac{1}{2^{8g}}\right)$. By limiting the number of detected faults to F_{max}, it is possible to use smaller values of g while still limiting the attacker's success probability $\Pr[\mathcal{A}_*^{1,F_{max}}]$ to a maximum value p_{max}. Accordingly, given F_{max} and p_{max}, a lower bound on g can be computed as:

$$g > \frac{-\log_2\left(1 - (1 - p_{max})^{1/F_{max}}\right)}{8} \text{ bytes} \tag{5}$$

Summary. Figure 2 and Figure 3 provide a summary of the security analysis. In particular, Figure 2 shows the $\log_2$ of the probability of a successful attack for increasing F_{max} values

[5]Faulting the root before its digest is computed naturally affects many elements of the digest. Nevertheless, for the purpose of exploitation, it suffices that any digest component is reduced below its expected value, as this already leads to an exploitable WOTS$^+$ signature. For simplicity, we therefore assume that a single digest element has been corrupted to a lower value.

and for different values of g. Figure 3 provides the value of the lower bound on g for increasing F_{max} values and for different values of the maximum probability of a successful attack p_{max}[6]. Accordingly, Figure 3 can be used to select the minimum value of g such that the probability of a successful attack is lower than p_{max} for F_{max} fault attempts/SLH-DSA signature generation executions. For instance, assuming a fault counter with a maximum value of 100 and a target maximum probability of attack success of 2^{-8}, the minimum integer value of g is 2 bytes. Another example, assuming no fault counter and a target maximum probability of attack success of 2^{-40}, a value of $g = 8$ bytes is sufficient to achieve the target attack resilience against an attacker who can request and fault 10^7 SLH-DSA signature generations.

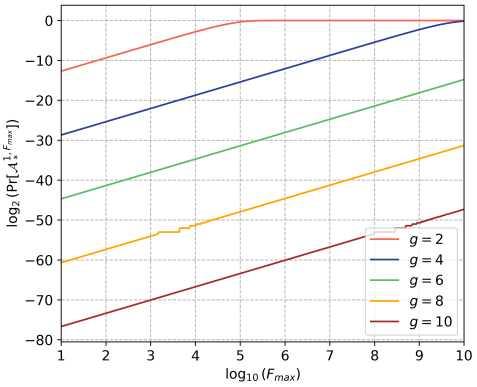


Figure 2: $\log_2(\Pr[\mathcal{A}_*^{1,F_{max}}])$ as function of F_{max} for different g values.

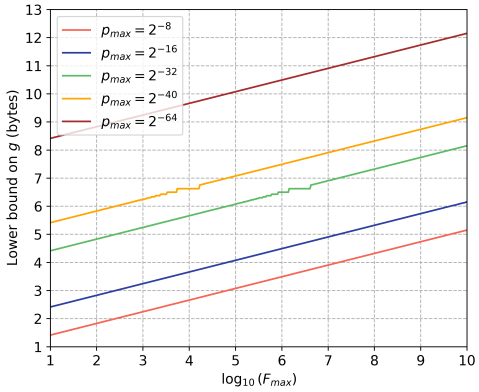


Figure 3: Lower bound on g as function of F_{max} for different p_{max} values.

4.3 Broader fault resilience

The compressed caching countermeasure, similarly to standard double computation provides resistance against one fault per execution, which is parameterizable as function of g as presented in the previous section. In the following we investigate whether compressed caching can provide resistance against two or more faults during a single execution of the signature generation. Again, we assume that faults are always succesfully injected, and additionally that the first fault in the two- or multiple-fault scenario is any arbitrary fault that leads to an exploitable WOTS$^+$ signature. We denote such a fault by $*$. The remaining faults are aiming to bypass the compressed caching countermeasure. Accordingly we investigate the following attack possibilities:

Random fault - $\mathcal{A}_{*,r}^{2,F}$ and $\mathcal{A}_{*,r,r,\ldots,r}^{m,F}$. For this attack, in addition to the fault inducing an exploitable WOTS$^+$ signature, an attacker able to only inject a random fault can aim to essentially "correct" the effect of the fault such that the fault is not detected in the comparison.

$$\tilde{X} = \overbrace{\texttt{wots_sign}(\mathbf{SK}, root)}^{\lightning *} \rightarrow \overbrace{\mathsf{Trunc}_g(\mathcal{H}(\tilde{X})) \rightarrow \boxed{=?} \leftarrow \{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{cache}}^{\lightning r}$$

Eventually, the best strategy is to target $\mathsf{Trunc}_g(\mathcal{H}(\tilde{X}))$ or $\{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{cache}$ (due to these values being limited to g bytes as opposed to the n-byte hash outputs) and the probability of success is then the same as for $\mathcal{A}_r^{1,F}$. The same holds for attacker $\mathcal{A}_{*,r,r,\ldots,r}^{m,F}$

[6]The step-like parts in the curves result from using trivial log and exp approximations to avoid overflow and domain errors when handling extremely large or small values.

since all faults are random.

$$\Pr[\mathcal{A}^{2,F}_{*,r}] = \Pr[\mathcal{A}^{m,F}_{*,r,r,\ldots,r}] = \Pr[\mathcal{A}^{1,F}_{r}] = 1 - \left(1 - \frac{1}{2^{8g}}\right)^F$$

Skipping fault - $\mathcal{A}^{2,F}_{*,s}$ and $\mathcal{A}^{m,F}_{*,s,s,\ldots,s}$. This attack path is quite promising for an attacker. They first inject an arbitrary fault leading to an exploitable faulty signature and then inject a second fault to skip or bypass the comparison.

$$\tilde{X} = \overbrace{\texttt{wots_sign}(\mathbf{SK}, root)}^{\lightning *} \rightarrow \mathsf{Trunc}_g(\mathcal{H}(\tilde{X})) \rightarrow \boxed{=?}^{\lightning s} \leftarrow \{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{\text{cache}}$$

Assuming perfect fault injection, the attacker succeeds and a faulty WOTS$^+$ signature is returned. This attack path, however, can be effectively mitigated with minimal impact by recomputing the comparison as many times as deemed necessary. Even with additionally recomputing the truncated hash prior to each comparison, the performance impact of this mitigation remains negligible, when contrasted with the large number of hash calls already required in a single SLH-DSA signature generation. The same mitigation can be employed to hinder attacker $\mathcal{A}^{m,F}_{*,s,s,\ldots,s}$ by recomputing the comparison $(m-1)$ times. We analyze the concrete impact in Section 6.3.

Controlled fault - $\mathcal{A}^{2,F}_{*,c}$ and $\mathcal{A}^{m,F}_{*,c,c,\ldots,c}$. The scenario of (multiple) controlled faults is extremely powerful, since we assume that, an attacker is able to modify an intermediate value to a value of their choosing. This attacker can also be generalized to an attacker who induces many targeted bit flips during the same execution, i.e., $\mathcal{A}^{m,F}_{*,c,c,\ldots,c}$, to change an intermediate value to a value of their choosing. Such a powerful attacker model and assumption may not be realistic in practice, however, we still consider them in this section to scrutinize the security of the proposed compressed caching countermeasure and offer efficient mitigations against such powerful attacker as well.

One attack strategy to bypass the comparison is to set $\mathsf{Trunc}_g(\mathcal{H}(\tilde{X}))$ to its expected value in the cache, namely $\{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{cache}$. This is possible, because the attacker has knowledge of $\mathsf{Trunc}_g(\mathcal{H}(X))$ simply by observing a valid SLH-DSA signature containing X. However, they do not know the value of $\mathsf{Trunc}_g(\mathcal{H}(\tilde{X}))$ without knowledge of the WOTS$^+$ secret key. Therefore, faulting $\{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{cache}$ to a chosen value does not improve the sucess probability of an attacker, i.e., it remains probabilitic as a function of g. Hence, under the attack assumptions we make in this section, the best strategy for a $\mathcal{A}^{2,F}_{*,c}$ attacker is to first inject an arbitrary fault leading to an exploitable faulty signature and then inject a second fault to set $\mathsf{Trunc}_g(\mathcal{H}(\tilde{X}))$ to $\mathsf{Trunc}_g(\mathcal{H}(X))$.

$$\tilde{X} = \overbrace{\texttt{wots_sign}(\mathbf{SK}, root)}^{\lightning *} \rightarrow \overbrace{\mathsf{Trunc}_g(\mathcal{H}(\tilde{X}))}^{\lightning c} \rightarrow \boxed{=?} \leftarrow \{\mathsf{Trunc}_g(\mathcal{H}(X))\}_{\text{cache}}$$

As before, assuming perfect fault injection the attacker succeeds and a faulty and exploitable WOTS$^+$ signature is returned. To mitigate this attack vector, it is required to recompute not only the comparison as in the skipping case, but the full computation of $\mathsf{Trunc}_g(\mathcal{H}(\tilde{X}))$. Note that, it is important that $\tilde{X}$ is written out as the signature only after all checks have passed. Otherwise, the attacker could fault $\tilde{X}$ once to impact all recomputations. Recall that the attacker does not know the value of $\tilde{X}$, since it depends on the WOTS$^+$ secret key. In effect, they cannot fault $\tilde{X}$ to a known value to bypass all recomputations, and then fault it back to its original value before it is written out. As before, the same mitigation can be employed to hinder attacker $\mathcal{A}^{m,F}_{*,c,c,\ldots,c}$ by recomputing the hash and comparison $(m-1)$ times, which still has negligible performance impact on the overall signature generation as shown in Section 6.3.

Summary. Figure 4 depicts the summary of our analysis regarding protection against attackers who can inject multiple faults per signature generation. One arbitrary fault during the WOTS$^+$ signature generation is always required to generate an exploitable faulty signature. Further faults are required to overcome the compressed caching comparison, which can be skipping faults or controlled faults (or a combination). To protect against these attack vectors introduced by the compressed caching countermeasure, we recompute the truncated hash and the comparison $(m-1)$ times. As depicted in Figure 4, it is essential that the WOTS$^+$ signature is only written out after all checks have passed, to avoid that an attacker bypasses the additional checks.

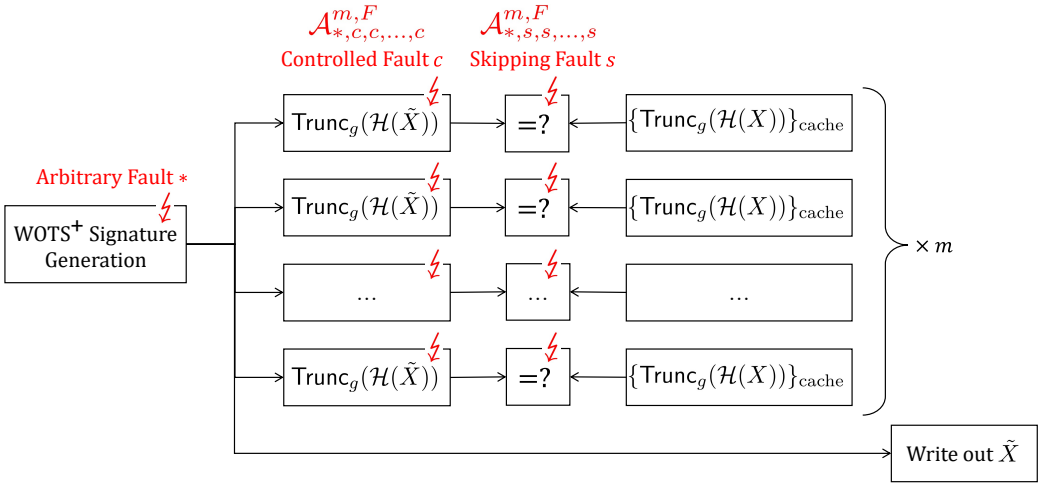


Figure 4: Multiple-fault attacks on the compressed caching countermeasure.

5 Performance Analysis of Compressed Caching

In this section, we consider that non-cached layers and any other hash calls not involved in the cached layers are recomputed to protect against faults. The total hash function calls in `slh_sign` protected with compressed caching the top c' layers (with $c' \in \{0, 1, 2, \ldots, d\}$ and $c' = 0$ means no caching) is equal to:

$$\#\text{Total}^S_{comp_caching}(c') = \begin{cases} \#\text{Total}^S_{recomp} & \text{if } c' = 0 \\ 2 \cdot \Big(\mathbf{PRF}_{msg} + \mathbf{H}_{msg} + \#\text{Total}^{FS} + \#\text{Total}^{Fpk} \\ \quad + (d - c') \cdot \#\text{Total}^{XS} + (d - c') \cdot \#\text{Total}^{Xpk} \Big) \\ \quad + c' \cdot \#\text{Total}^{XS} + (c' - 1) \cdot \#\text{Total}^{Xpk} & \text{if } c' \geq 1 \end{cases}$$

The blue part denotes the operations that are not recomputed due to compressed caching. The memory required in bytes to cache the compressed WOTS$^+$ data for c' layers is:

$$\#\text{Mem}^S_{comp_caching}(c', g) = \frac{2^{h'}\left(2^{c'h'} - 1\right)}{2^{h'} - 1} g \text{ bytes} \tag{6}$$

Memory requirements comparison. First, we simply compare the total memory needed to cache the same number of layers as standard caching, i.e., $c' = c$. Table 4 provides the memory needed in bytes for the compressed caching for an increasing number of layers for

$g = 4$ bytes. The rightmost column of Table 4 titled 'Memory reduction' corresponds to the % of memory saved compared to the standard caching. It is significant since compressed caching uses considerably less memory than standard caching. Eventually, taking into account Equation (1) and Equation (6) the memory reduction factor when $c' = c$ is $\frac{g}{(\ell+1)\cdot n}$, which is significant even for larger values of g.

Table 4: Memory required for the SLH-DSA compressed caching countermeasure with $g = 4$ bytes.

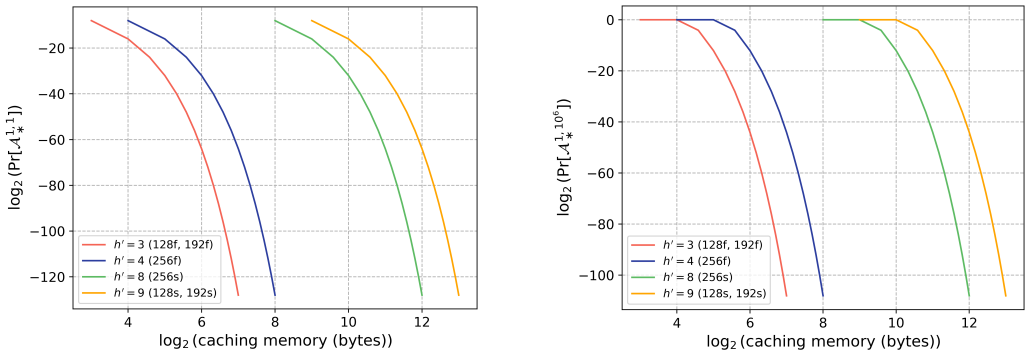
Parameter set	**Memory (bytes)** $c' = 1$	$c' = 2$	$c' = 3$	$c' = 4$	...	$c' = d$	**Memory reduction**
SHA2/SHAKE-128s	2048	1.05×10^6	5.38×10^8	2.75×10^{11}	...	3.70×10^{19}	99.3 %
SHA2/SHAKE-128f	32	288	2336	1.87×10^4	...	3.37×10^{20}	99.3 %
SHA2/SHAKE-192s	2048	1.05×10^6	5.38×10^8	2.75×10^{11}	...	3.70×10^{19}	99.7 %
SHA2/SHAKE-192f	32	288	2336	1.87×10^4	...	3.37×10^{20}	99.7 %
SHA2/SHAKE-256s	1024	2.63×10^5	6.74×10^7	1.72×10^{10}	...	7.41×10^{19}	99.8 %
SHA2/SHAKE-256f	64	1088	1.75×10^4	2.80×10^5	...	1.26×10^{21}	99.8 %

Caching the full WOTS$^+$ signatures and public keys for c layers provides a higher performance gain, in terms of hash calls, compared to using compressed caching for $c' = c$ layers. At the end of this section, we show that due to the significant memory savings offered by compressed caching, we can cache more layers using compressed caching, i.e., $c' > c$ and achieve better performance than standard caching for less required memory.

Security vs. memory trade-off. Since g impacts both the fault resistance that can be achieved and the memory required by compressed caching, we next analyze the security to memory trade-off. In Figure 5, we plot on the y-axis the $\log_2$ of the attack success probability as function of the $\log_2$ of the required caching memory in bytes. Memory increases as g increases, whereas the probability of a successful attack decreases with g. The plot is provided for caching one layer, i.e., $c' = 1$. Each additional layer will shift the values on the x-axis by $\approx h'$ (as clear from Equation 6). Figure 5a provides the trade-off plot for an attacker with $F = 1$ faulted signature generations and Figure 5b provides the trade-off plot for an attacker with $F = 10^6$ faulted signature generations. The comparison between the two plots highlights as expected that a higher value of g, and accordingly more memory, is required to lower the probability of a successful attack for $F = 10^6$ compared to $F = 1$. However, eventually with a high enough value of g, which accordingly makes the term $\frac{1}{2^{8g}}$ in the attack success probability given in Equation 4 very small, high security can also be achieved for $F = 10^6$.

Number of hash calls comparison. Compared to standard caching, compressed caching offers a finer-grained trade-off between fault resilience, memory (or cache size) and performance. In the following, let M_{max} be the maximum memory (in bytes) available for caching, either standard or compressed. The goal is to find $c' \geq c$ s.t. both the memory used by standard caching and compressed caching are below M_{max}. Then, given c for standard caching and c' for compressed caching, we analyze the performance improvement over a full recomputation of the signature generation, in terms of number of hash calls. One hash call corresponds to one call to one of the hash or pseudorandom functions given in Table 1[7]. We provide results for $g \in \{2, 8\}$ corresponding to the two exemplary values of g mentioned in Section 4 and $M_{max} \in \{2\text{ kB}, 16\text{ kB}, 128\text{ kB}\}$ for illustration purposes.

[7]For the SHAKE parameter sets, we also computed the number of Keccak calls for the different hash and pseudorandom functions, including the hash function used to compress the WOTS$^+$ data for compressed caching. These functions naturally have a varying number of Keccak calls depending on the input and



(a) $F = 1$ (b) $F = 10^6$

Figure 5: Security vs. memory trade-off of the compressed caching countermeasure.

Note that such sizes also mean that the full cache can be loaded in RAM on many different target platforms[8]. An overview of the results is given in Table 5 and Table 6 for $g = 2$ bytes and $g = 8$ bytes, respectively.

First, note that depending on the parameter set and M_{max}, it may be impossible to cache the WOTS$^+$ and public keys for a full layer. A similar observation holds for compressed caching, albeit for larger values of g and smaller values of M_{max} (see the annotated a in Table 6). Caching the full top layer can be impossible for standard caching even for large values of M_{max}, mainly for the small (s) parameter sets for which h' is larger compared to the fast (f) variants (see the annotated b in Table 5 and Table 6). In such cases, i.e., for the small SLH-DSA parameter sets or for relatively small M_{max}, compressed caching can make use of a few bytes of memory to achieve a slight performance improvement compared to recomputation. For instance, for the 256s parameter set, compressed caching can use 512 bytes when $g = 2$ and ≈ 2.1 kB when $g = 8$ to achieve a 4.2% improvement in the number of hash calls. In general, for the small parameter sets caching the full top layer can already be prohibitive in terms of memory since it contains $2^{h'}$ WOTS$^+$ instances with $h' \in \{8, 9\}$.

For the fast parameter sets, more options and trade-offs are available for caching full layers using compressed caching. We notice that in some cases compressed caching can use the available memory more efficiently to achieve a better speed-up compared to standard caching (see a few examples annotated by c in Table 5 and Table 6). Naturally, as highlighted in the previous section, compressed caching can be used to significantly reduce the memory required to cache the same number of layers as for standard caching. If the goal of the implementer is to reduce memory requirements drastically, then this can be achieved by compressed caching for some performance degradation compared to standard caching (see a few examples annotated by d in Table 5 and Table 6). However, one interesting observation from Table 5 and Table 6 is that in some cases compressed caching can either achieve comparable performance improvements to standard caching for a fraction of the required memory or achieve a, albeit marginally better, performance improvement compared to standard caching for less caching memory (see a few examples annotated by e in Table 5 and Table 6).

output sizes. We found that this had virtually no impact on the relative performance improvement percentages compared to counting hash calls. We hence provide results using hash calls as a metric for generality.

[8]For instance, most of the micro-controller boards used by the PQM4 benchmarking framework https://github.com/mupq/pqm4

Table 5: Comparison of standard caching and compressed caching for $g = 2$ bytes. The memory improvement given in () corresponds to the % memory saved compared to standard caching. The hash call improvements given in () are relative to the number of hash calls required for a SLH-DSA signature generation protected with double computation.

Parameter set	c	**Memory**	**Hash calls**	c'	**Memory**	**Hash calls**
M_{max}	**2 kB**					
SHA2/SHAKE-128s	0	0	4 372 k (0%)	1	1 kB	4 084 k (6.6%)
SHA2/SHAKE-128f	0	0	210 k (0%)	1	16 B	206 k (2%)
				2	144 B	201 k (4.2%)
				3	1.2 kB	197 k (6.3%)
SHA2/SHAKE-192s	0	0	7 534 k (0%)	1	1 kB	7 115 k (5.5%)
SHA2/SHAKE-192f	0	0	338 k (0%)	1	16 B	332 k (1.8%)
				2	144 B	325 k (3.8%)
				3	1.2 kB	318 k (5.7%)
SHA2/SHAKE-256s	0	0	6 561 k (0%)	1	512 B	6 286 k (4.2%)
SHA2/SHAKE-256f	0	0	691 k	1	32 B	674 k (2.4%)
				2	544 B	657 k (4.9%)
M_{max}	**16 kB**					
SHA2/SHAKE-128s	0	0	4 372 k (0%)	1	1 kB	4 084 k (6.6%)
SHA2/SHAKE-128f	1	4.6 kB	201 k (4%)	1^d	16 B (99.6%)	206 k (2%)
				2^e	144 B (96.9%)	201 k (4.2%)
				3^c	1.2 kB (74.6%)	197 k (6.3%)
				4	9.4 kB (-103%)	192 k (8.4%)
SHA2/SHAKE-192s	0	0	7 534 k (0%)	1	1 kB	7 115 k (5.5%)
SHA2/SHAKE-192f	1	10 kB	325 k (3.6%)	1^d	16 B (99.8%)	332 k (1.8%)
				2^e	144 B (98.6%)	325 k (3.8%)
				3^e	1.2 kB (88.2%)	318 k (5.7%)
				$4^{c,e}$	9.4 kB (6.2%)	312 k (7.6%)
SHA2/SHAKE-256s	0	0	6 561 k (0%)	1	512 B	6 286 k (4.2%)
SHA2/SHAKE-256f	0	0	691 k	1	32 B	674 k (2.4%)
				2	544 B	657 k (4.9%)
				3	8.7 kB	640 k (7.4%)
M_{max}	**128 kB**					
SHA2/SHAKE-128s	0^b	0	4 372 k (0%)	1	1 kB	4 084 k (6.6%)
SHA2/SHAKE-128f	2	41.5 kB	192 k (8.3%)	1	16 B (99.9%)	206 k (2%)
				2	144 B (99.6%)	201 k (4.2%)
				3^d	1.2 kB (97.2%)	197 k (6.3%)
				4^e	9.4 kB (77.4%)	192 k (8.4%)
				5	74.9 kB (-80.6%)	188 k (10.6%)
SHA2/SHAKE-192s	0^b	0	7 534 k (0%)	1	1 kB	7 115 k (5.5%)
SHA2/SHAKE-192f	2	89.9 kB	312 k (7.5%)	1	16 B (99.9%)	332 k (1.8%)
				2	144 B (99.8%)	325 k (3.8%)
				3	1.2 kB (98.7%)	318 k (5.7%)
				4^e	9.4 kB (89.6%)	312 k (7.6%)
				$5^{c,e}$	74.9 kB (16.6%)	305 k (9.6%)
SHA2/SHAKE-256s	0^b	0	6 561 k (0%)	1	512 B	6 286 k (4.2%)
SHA2/SHAKE-256f	1	34.8 kB	657 k (4.8%)	1	32 B (99.9%)	674 k (2.4%)
				2^e	544 B (98.4%)	657 k (4.9%)
				3^e	8.7 kB (74.9%)	640 k (7.4%)

Table 6: Comparison of standard caching and compressed caching for $g = 8$ bytes. The memory improvement given in () corresponds to the % memory saved compared to standard caching. The hash call improvements given in () are relative to the number of hash calls required for a SLH-DSA signature generation protected with double computation.

Parameter set	c	**Memory**	**Hash calls**	c'	**Memory**	**Hash calls**
M_{max}	**2 kB**					
SHA2/SHAKE-128s	0	0	4 372 k (0%)	0^a	0	4 372 k (0%)
SHA2/SHAKE-128f	0	0	210 k (0%)	1	64 B	206 k (2%)
				2	576 B	201 k (4.2%)
SHA2/SHAKE-192s	0	0	7 534 k (0%)	0	0	7 534 k (0%)
SHA2/SHAKE-192f	0	0	338 k (0%)	1	64 B	332 k (1.8%)
				2	576 B	325 k (3.8%)
SHA2/SHAKE-256s	0	0	6 561 k (0%)	0	0	6 561 k (0%)
SHA2/SHAKE-256f	0	0	691 k	1	128 B	674 k (2.4%)
M_{max}	**16 kB**					
SHA2/SHAKE-128s	0	0	4 372 k (0%)	1	4.1 kB	4 084 k (6.6%)
SHA2/SHAKE-128f	1	4.6 kB	201 k (4%)	1	64 B (98.6%)	206 k (2%)
				2^e	576 B (87.5%)	201 k (4.2%)
				3	4.7 kB (-1.4%)	197 k (6.3%)
SHA2/SHAKE-192s	0	0	7 534 k (0%)	1	4.1 kB	7 115 k (5.5%)
SHA2/SHAKE-192f	1	10 kB	325 k (3.6%)	1^d	64 B (99.4%)	332 k (1.8%)
				2^e	576 B (94.2%)	325 k (3.8%)
				$3^{c,e}$	4.7 kB (53.2%)	318 k (5.7%)
SHA2/SHAKE-256s	0	0	6 561 k (0%)	1	2.1 kB	6 286 k (4.2%)
SHA2/SHAKE-256f	0	0	691 k (0%)	1	128 B	674 k (2.4%)
				2	2.2 kB	657 k (4.9%)
M_{max}	**128 kB**					
SHA2/SHAKE-128s	0^b	0	4 372 k (0%)	1	4.1 kB	4 084 k (6.6%)
SHA2/SHAKE-128f	2	41.5 kB	192 k (8.3%)	1	64 B (99.8%)	206 k (2%)
				2	576 B (98.6%)	201 k (4.2%)
				3	4.7 kB (88.7%)	197 k (6.3%)
				4^e	37.4 kB (9.7 %)	192 k (8.4%)
SHA2/SHAKE-192s	0^b	0	7 534 k (0%)	1	4.1 kB	7 115 k (5.5%)
SHA2/SHAKE-192f	2	89.9 kB	312 k (7.5%)	1	64 B (99.9%)	332 k (1.8%)
				2	576 B (99.3%)	325 k (3.8%)
				3	4.7 kB (94.8%)	318 k (5.7%)
				4^e	37.4 kB (58.3%)	312 k (7.6%)
SHA2/SHAKE-256s	0^b	0	6 561 k (0%)	1	2.1 kB	6 286 k (4.2%)
SHA2/SHAKE-256f	1	34.8 kB	657 k (4.8%)	1^d	128 B (99.6%)	674 k (2.4%)
				2^e	2.2 kB (93.7%)	657 k (4.9%)
				3	34.9 kB (-0.4%)	640 k (7.4%)

Summary. To summarize, compressed caching can provide some performance improvement in terms of total hash calls compared to recomputation and standard caching for a small (for instance, relative to the signature size) amount of memory for the small parameter sets or when only limited (e.g., $\leq$ 2kB) caching memory is available for small and fast parameter sets. For instance for the small parameter sets, the performance improvement can be of up 6.6% and when 2 kB or less of caching memory are available, the improvement can be of up to 6.3% and up to 4.2% for $g = 2$ bytes and $g = 8$ bytes, respectively. It can also provide comparable performance to standard caching but for a fraction of the memory used by standard caching. Depending on the parameter set, M_{max} and g, the memory improvement for comparable performance can be of up to 98.5%.

Compressed caching also offers a better trade-off between caching memory and the number of hash calls for the fast parameter sets. For instance, an up to 8.4% improvement in terms of total hash calls during signing for less than 10 kB of caching memory when $g = 2$ bytes and less than 40kB when $g = 8$ bytes. In some cases, compressed caching can achieve a small improvement in the number of hash calls for a negligible (relative to the SLH-DSA signature size and other intermediates during signing) amount of memory, namely $\approx 2\%$ reduction in the number of hash calls for less than a few hundreds bytes of memory.

6 Proof-of-Concept Implementation and Validation

Using the equations provided in this paper, the practical performance improvement can be closely estimated by substituting the costs of the various hash and pseudorandom function types (since the majority of SLH-DSA's cost lies in hashing). Nonetheless, we include the results in this section to confirm that the estimated gains are indeed reflected in practical implementations. We also use this section to validate the fault detection probability.

6.1 Implementation

In this section, we provide some performance numbers on a standard laptop. While so far we investigated improvements in terms of hash calls, the goal of this section and this proof of concept is to assess whether the theoretical performance improvements provided in Section 5 are accurate when taking clock cycles into account. We use the SPHINCS$^+$ (SLH-DSA) reference code[9] and update it to support standard caching and compressed caching, along with recomputation. For both kinds of caching, the cache is filled after key generation but before signing. This task can be time consuming depending on the values of c, c' and h'. However, the cache is fixed for each key pair and hence this is only done once per key pair, e.g., during key generation or before signing the first message. For the signature generation routines using caching, we recompute and compare the FORS signatures and the XMSS signatures for all non-cached layers, whereas for the recomputation-only signature generation, we recompute the full SLH-DSA signature and compare.

We provide results for values of c and c' accommodating a maximum cache size of 128kB in Table 7. Specifically, for compressed caching we use $g = 2$ bytes, simply so it is easy to put the bottom of Table 5 and Table 7 side-by-side for comparison and have more options for c' to compare theoretical and practical performance improvement numbers. In any case, the value of g naturally impacts the total memory used by compressed caching but has a negligible impact on performance since it only impacts c' hash calls (namely, only the requested output size of the XOF) out of the many hash calls during signature generation. Memory used by caching for the different values of c and c' is unchanged and is provided in Table 5.

[9] https://github.com/sphincs/sphincsplus/tree/consistent-basew

We average the clock cycles over 100 random signatures for the small variants and over 1000 random signatures for the fast variants. Then for each batch of signatures, we estimate the improvement over a full recomputation. Improvements are given in (%). The number of hash calls and hence the clock cycles are highly variable, since each new random message leads to a different path in the hypertree and hence different numbers of hash calls during the WOTS$^+$ signing steps. Therefore, we provide in the recomputation column the average for the same batch of messages used to estimate the improvements provided by standard caching and compressed caching.

Table 7: Average clock cycles for standard caching, compressed caching and full recomputation. Improvement compared to full recomputation are provided in (%).

Parameter set	**Standard caching**		**Compressed caching**		**Recomputation**
	c	**Clock cycles**	c'	**Clock cycles**	**Clock cycles**
SHAKE-128s	0	4.85×10^9 (0%)	1	4.53×10^9 (6.75%)	4.85×10^9
SHAKE-128f	1	2.10×10^8 (2.57%)	1	2.12×10^8 (1.9%)	2.16×10^8
	2	2.05×10^8 (8.63%)	2	2.17×10^8 (3.62%)	2.25×10^8
			3	2.04×10^8 (6.53%)	2.18×10^8
			4	1.99×10^8 (8.65%)	2.18×10^8
			5	1.93×10^8 (11.81%)	2.19×10^8
SHAKE-192s	0	8.10×10^9 (0%)	1	7.72×10^9 (4.74%)	8.10×10^9
SHAKE-192f	1	3.38×10^8 (3.17%)	1	3.44×10^8 (1.22%)	3.49×10^8
	2	3.33×10^8 (6.9%)	2	3.45×10^8 (3.68%)	3.58×10^8
			3	3.25×10^8 (6.69%)	3.48×10^8
			4	3.18×10^8 (7.09%)	3.42×10^8
			5	3.05×10^8 (9.71%)	3.38×10^8
SHAKE-256s	0	6.72×10^9(0%)	1	6.43×10^9 (4.26%)	6.72×10^9
SHAKE-256f	1	6.75×10^8 (4.41%)	1	6.90×10^8 (2.41%)	7.07×10^8
			2	6.65×10^8 (4.79%)	6.99×10^8
			3	6.60×10^8 (7.13%)	7.10×10^8

The results provided in Table 7 confirm the assessment from Tables 5 and Tables 6. This is somewhat expected as the bulk of SLH-DSA operations is hashing. However, these assessments assume dealing with the cache, mainly reading from the cache, is inexpensive as it can be fully loaded in RAM, e.g., before signing a batch of messages. For more memory constrained devices, since compressed caching can be parameterized to use a fraction of the memory that standard caching uses, it can still use a very small amount of memory to provide an also small performance improvement. There may also be strategies to allow using a larger cache efficiently with limited RAM. We leave the investigation of such strategies as a direction for future work.

6.2 Fault simulation

In this section, we simulate a worst-case attacker able to fault the exact intermediates that lead to a compromised WOTS$^+$ signature. To do so we modified the signing procedure to take into account the following fault models and inject random faults ($f() \neq id()$) in one of the corresponding intermediates.

- `FAULT_ROOT`: the root to be signed by a WOTS$^+$ secret key is corrupted.
- `FAULT_CHAINS`: the number of hashes in a WOTS$^+$ chain value is corrupted.
- `FAULT_AUTH_PATH`: one of the leaves at the bottom of the target tree is corrupted.

For each kind of fault we estimate the probability that the compressed caching countermeasure does **not** detect the fault, i.e., a detection miss rate. The goal of this experiment

is to confirm whether our security analysis, which makes an implicit assumption of randomness and uniformity because of the use of hash functions, and results from Section 4 hold. For that purpose, we use the SHAKE-128f parameter set and $g = 1$ byte to allow estimating the probabilities over a large number of signatures and get meaningful results[10]. We generate 100 SLH-DSA random key pairs and for each key pair generate 500 signatures. We use $c' \in \{1, 2, 3\}$ and always target the first cached layer. We would expect however similar results across all layers[11].

In addition to the previous probability, we also estimate the probability that the returned signature can be verified. As noted in [Gen23] verification after signing is not a suitable countermeasure against the grafting tree attack, as it does not detect all faults. In particular, there is a subset of values in the authentication path computation, that when corrupted will still lead to a verifiable signature. For more details on the verifiability of faulted signatures, we refer the reader to the analysis provided by Genêt [Gen23].

Table 8: Perfect fault simulation results for the SLH-DSA-SHAKE-128f parameter set.

c'	Theoretical miss Pr	`FAULT_ROOT`		`FAULT_CHAINS`		`FAULT_AUTH_PATH`	
		Estimated miss Pr	Estimated verify Pr	Estimated miss Pr	Estimated verify Pr	Estimated miss Pr	Estimated verify Pr
1	≈ 0.003906	0.003920	0	0.003980	0	Not exploitable	
2	≈ 0.003906	0.004000	0	0.004060	0	0.003620	0.750900
3	≈ 0.003906	0.003700	0	0.003600	0	0.003740	0.750960

The results are provided in Table 8. The column titled 'Theoretical miss Pr' corresponds to the theoretical probability that a fault is undetected by compressed caching, i.e., $\frac{1}{2^8}$. The 'Estimated miss Pr' and 'Estimated verify Pr' columns correspond to the probability that the fault is undetected by the compressed caching and the probability that the faulty signature is verified despite the fault, respectively. In all fault model cases, the compressed caching provides a detection rate close to the theoretical value. This confirms that the security analysis provided in Section 4 can be used to set the value of g to parameterize the compressed caching countermeasure to provide resilience against faults up to a probability set by the implementer. For `FAULT_AUTH_PATH`, as previously mentioned, some faulty signatures still pass verification. In this case, $\approx 75\%$ of the faulty signatures are validated by verification, whereas compressed caching is able to detect almost all faulty signatures, up to its detection rate (as function of g).

6.3 Impact of mitigating multiple faults

As discussed in Section 4.3, mitigating multiple faults may require recomputing both the truncated hash of the WOTS$^+$ signature and the comparison to its corresponding value in the cache. Figure 6 shows the impact or overhead (in % of total clock cycles) incured by recomputing the truncated hash and the comparison numerous times. The overhead is relative to the clock cycles of computing the signature generation protected with compressed caching, i.e., by recomputing the bottom layers and leveraging the compressed caching countermeasure for the top c' layers. The x-axis corresponds to the number of times this recomputation is done for each single layer. Effectively, the truncated hash and comparison are computed a total of $(x + 1)$ times for each cached layer. We provide estimates for two parameter sets, namely for 128f and 128s. Since 128s is not as caching

[10] For a large g value, the probability of a fault not being detected is extremely low, which would require us to sample a significantly larger number of signatures.

[11] Some discrepancies are to be expected since for small h' and c', and for a fixed key, the set of sampled (through signing) WOTS$^+$, root or tree node values are not perfectly uniform over the set of all possible values.

friendly, the compressed caching is parametrized with $c' = 1$, whereas for 128f, we set $c' = 5$ to maximize the number of times a recomputation of a truncated hash and comparison is done.

It is clear from Figure 6 that recomputing the hash and the comparison multiple times has only a negligible cost compared to the overall cost of a compressed-caching-protected SLH-DSA signature generation. Performing the hash recomputation and comparison 100 times accounts for well under 1% of the overall clock cycles, at just above 0.2%. This is somewhat expected as signing with SLH-DSA requires hundreds of thousands to millions of hash calls, hence a few additional hash computations and comparisons do not induce a visible performance impact.

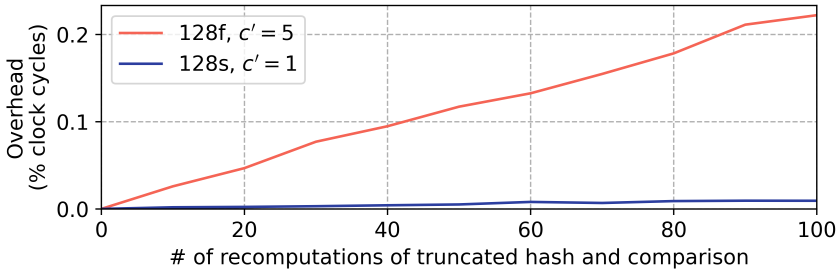


Figure 6: Overhead of recomputing the truncated hash and comparison many times compared to the overall cost of compressed-caching-protected signature generation.

7 Granular Caching - or when to use compressed caching

So far we investigated caching full layers. Naturally, one can use the full available memory M_{max} to cache as many pairs of WOTS$^+$ signatures and public keys (for standard caching) or truncated hashes of WOTS$^+$ signatures (for compressed caching) as possible. In the following, we provide the impact on the total number of hash calls for standard caching and compressed caching, when the available memory is used to cache as many layers as possible (c and c' layers for standard and compressed caching, respectively) and then the remaining memory is used to cache as many elements of the layer below as possible (e and e' elements for standard and compressed caching, respectively). We assume that for the incomplete layers, the WOTS$^+$ signatures are cached in order based on the tree and WOTS$^+$ key pair address.

Below we provide the equations to compute c, c', e, e' and hash counts for granular caching. We mainly use the granular versions of the caching strategies as a way to analyze the performance-memory trade-off on a finer-grained scale. Most importantly, we use granular caching to answer the question "when to use compressed caching instead of standard caching?".

Granular standard caching. Given M_{max} bytes of caching memory, with standard caching one can cache $c \leq d$ full layers and e individual WOTS$^+$ signatures and public keys on the subsequent layer, if and only if $c \leq d - 1$, with c and e defined as:

$$c = \left\lfloor \frac{1}{h'} \log_2 \left(\frac{(2^{h'} - 1)M_{max}}{2^{h'}(\ell + 1)n} + 1 \right) \right\rfloor \; ; \; e = \left\lfloor \frac{M_{max}}{(\ell + 1)n} - \frac{2^{h'} \left(2^{ch'} - 1 \right)}{2^{h'} - 1} \right\rfloor$$

With $\frac{e}{2^{(c+1)h'}}$ being the probability of encountering during signature generation a cached WOTS$^+$ signature for the non-fully cached layer, the average number of hash calls is:

$$\#\text{Total}^S_{gr_caching}(c,e) = \begin{cases} 2\cdot\Big(\mathbf{PRF}_{msg}+\mathbf{H}_{msg}+\#\text{Total}^{FS}+\#\text{Total}^{Fpk} \\ +\Big(d-\frac{e}{2^{h'}}\Big)\cdot\#\text{Total}^{XS}+(d-1)\cdot\#\text{Total}^{Xpk}\Big) \\ +\frac{e}{2^{h'}}\cdot\#\widetilde{\text{Total}^{XS}} \qquad \text{if } c=0 \\ \\ 2\cdot\Big(\mathbf{PRF}_{msg}+\mathbf{H}_{msg}+\#\text{Total}^{FS}+\#\text{Total}^{Fpk} \\ +\Big(d-c-\frac{e}{2^{(c+1)h'}}\Big)\cdot\Big(\#\text{Total}^{XS}+\#\text{Total}^{Xpk}\Big)\Big) \\ +\Big(c+\frac{e}{2^{(c+1)h'}}\Big)\cdot\#\widetilde{\text{Total}^{XS}} \qquad \text{if } c\geq 1 \end{cases}$$

where the blue parts denote the operations for the cached layers or elements and hence not recomputed.

Granular compressed caching. Given M_{max} bytes of caching memory, with compressed caching one can cache $c' \leq d$ full layers and e' individual truncated hashes of WOTS$^+$ signatures and public keys on the subsequent layer, if and only if $c' \leq d-1$, with c' and e' defined as:

$$c' = \left\lfloor \frac{1}{h'}\log_2\left(\frac{(2^{h'}-1)M_{max})}{2^{h'}g}+1\right)\right\rfloor \;;\; e' = \left\lfloor \frac{M_{max}}{g} - \frac{2^{h'}\left(2^{c'h'}-1\right)}{2^{h'}-1}\right\rfloor$$

Similarly, $\frac{e'}{2^{(c'+1)h'}}$ is the probability of encountering during signature generation a cached WOTS$^+$ hash for the non-fully cached layer, the average number of hash calls is then:

$$\#\text{Total}^S_{gr_comp_caching}(c',e') = \begin{cases} 2\cdot\Big(\mathbf{PRF}_{msg}+\mathbf{H}_{msg}+\#\text{Total}^{FS}+\#\text{Total}^{Fpk} \\ +\Big(d-\frac{e'}{2^{h'}}\Big)\cdot\#\text{Total}^{XS}+(d-1)\cdot\#\text{Total}^{Xpk}\Big) \\ +\frac{e'}{2^{h'}}\cdot\#\text{Total}^{XS} \qquad \text{if } c'=0 \\ \\ 2\cdot\Big(\mathbf{PRF}_{msg}+\mathbf{H}_{msg}+\#\text{Total}^{FS}+\#\text{Total}^{Fpk} \\ +\Big(d-c'-\frac{e'}{2^{(c'+1)h'}}\Big)\cdot\Big(\#\text{Total}^{XS}+\#\text{Total}^{Xpk}\Big)\Big) \\ +\Big(c'+\frac{e'}{2^{(c'+1)h'}}\Big)\cdot\#\text{Total}^{XS} \\ +\Big(c'+\frac{e'}{2^{(c'+1)h'}}-1\Big)\cdot\#\text{Total}^{Xpk} \qquad \text{if } c'\geq 1 \end{cases}$$

where the blue parts denote the operations for the cached layers or elements and hence not recomputed.

 # hash calls vs. memory trade-off analysis. In this section, we analyze the time-memory trade-off provided by (granular) standard caching and (granular) compressed caching. For

this purpose, we plot on Figure 7 and Figure 8, the logarithm of the average number of hash calls needed to generate a signature (protected by caching and recomputing the non-cached layers/XMSS signatures) as function of the logarithm of the caching memory in bytes, for small and fast parameter sets, respectively. Hence, lower is better. On both sets of figures, for (granular) compressed caching, the left hand side corresponds to $g = 2$ bytes and the right hand side to $g = 8$ bytes. On all figures, the continuous lines correspond to granular standard caching, where the • points correspond to the non-granular configurations ($e = 0$). The dashed lines correspond to granular compressed caching, where the + points correspond to the non-granular configurations ($e' = 0$).

First, note that increasing g simply shifts all the compressed caching curves (dashed) to the right, as they simply use more memory. What the figures mainly show is the regions where compressed caching is useful. In a nutshell, for low caching memory ($<\approx 2^{10}$ bytes), compressed caching offers the option of caching some data to speed up signing, whereas caching a single WOTS$^+$ signature is either not possible due to its size or caching memory is too limited to cache enough WOTS$^+$ signatures to lead to faster signing compared to compressed caching. This is made clear by the continuous curves only starting when at least $(\ell + 1)n$ bytes of caching memory are available and the dashed curves being below the continuous curves. For moderate caching memory ($\in \approx [2^{10}, 2^{18}]$ bytes), compressed caching offers a better hash-calls-to-memory trade-off than standard caching.

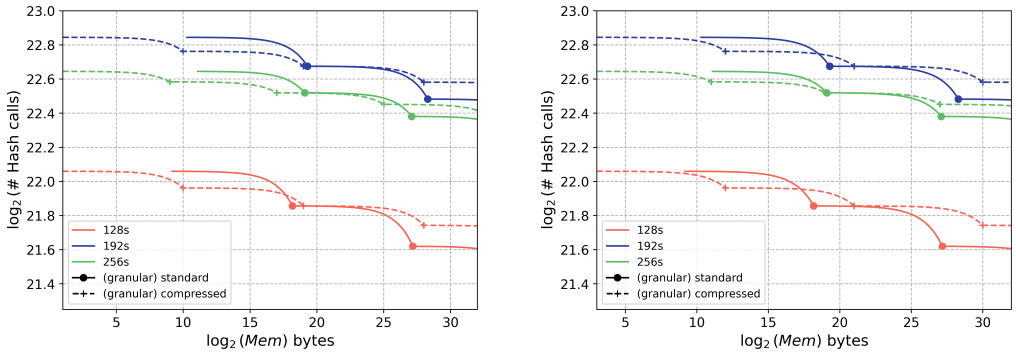


Figure 7: $\log_2$ # hash calls as function of $\log_2$ memory used for caching for small (s) SLH-DSA parameter sets. For compressed caching, $g = 2$ bytes on the left and $g = 8$ bytes on the right.

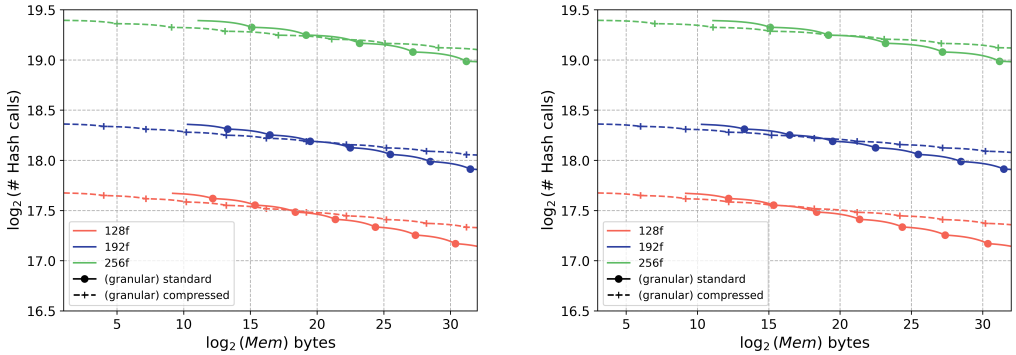


Figure 8: $\log_2$ # hash calls as function of $\log_2$ memory used for caching for fast (f) SLH-DSA parameter sets. For compressed caching, $g = 2$ bytes on the left and $g = 8$ bytes on the right.

Given a caching memory of more than $\approx 2^{18}$ bytes (i.e., 256 kB), the dashed and continuous curves intersect signaling that when enough memory is available, standard caching offers a better trade-off than compressed caching. This means that for most embedded or constrained devices, compressed caching offers a better use of caching memory. These are the types of devices for which fault attacks and optimized fault-hardened signing are the most relevant. For devices or applications with the possibility to cache a large amount of data, standard caching is the better option. The intersection point naturally changes based on the parameter set and the value of g, however it seems that g has a limited impact on this. For practical applications, given the parameter set and g the exact intersection point can be computed from the equations provided in this paper or by adapting the equations provided to the specific implementation.

Regarding granular vs. non-granular caching, while granular caching offers a way to use a maximum memory budget to achieve an additional performance improvement, non-granular caching is naturally easier to implement and provides a uniform execution and fault resistance behaviour for all WOTS$^+$ keys belonging to the same layer in the hypertree.

8 Conclusion

In this paper, we propose and analyze a novel and alternative approach to protecting SLH-DSA implementations against grafting tree attacks. We show that our so-called compressed caching countermeasure can be parameterized to provide very strong resilience against faults and use a fraction of the memory required by standard caching of the WOTS$^+$ signatures and public keys [Gen23]. In addition, it offers better performance-memory trade-offs when low to moderate caching memory is available, i.e., memory sizes that are most relevant for embedded devices. Only when over $\approx$ 256 kB of caching memory are available, is it outperformed by standard caching.

The introduced compressed caching is a new approach to protecting SLH-DSA implementations against faults and hence offers multiple avenues for further research. First, as mentioned in the paper, since compressed caching is relevant for embedded and more constrained devices, one interesting aspect to explore is the impact of limited runtime memory on compressed caching, or generally caching countermeasures for SLH-DSA, and potentially strategies to deal with very limited runtime memory. Second, we've analyzed compressed caching with software implementations in mind. One open question remains for both standard and compressed caching as to whether their performance improvements over recomputation are also translated to hardware optimized implementations, such as proposed in [Saa24].

Acknowledgments.

This result is part of the IPCEI ME/CT and is funded by the European Union Next Generation EU, the German Federal Ministry for Economic Affairs and Energy, the Bavarian Ministry of Economic Affairs, Regional Development and Energy, the Free State of Saxony with the help of tax revenue based on the budget approved by the Saxon State parliament and the Free and Hanseatic City of Hamburg.

This work is part of the SINFONIA project which has received funding from the Recovery and Resilience Facility (RRF) as the centrepiece of NextGenerationEU via the Austrian Research Promotion Agency (FFG) and Austria Wirtschaftsservice Gesellschaft mbH (aws) in the frame of the IPCEI ME/CT – Important Project of Common European Interest on Microelectronic and Communication Technologies under FFG project No 917423 and AWS project No P2431566.

References

[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.

[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, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2129–2146. ACM, 2019.

[CML+11] Gaetan Canivet, Paolo Maistri, Régis Leveugle, Jessy Clédière, Florent Valette, and Marc Renaudin. Glitch and laser fault attacks onto a secure AES implementation on a sram-based FPGA. *J. Cryptol.*, 24(2):247–268, 2011.

[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, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 165–184. Springer, 2018.

[Gen23] Aymeric Genêt. On protecting SPHINCS+ against fault attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):80–114, 2023.

[GKPM18] Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on SPHINCS. *IACR Cryptol. ePrint Arch.*, page 674, 2018.

[GOW+24] Ivan Gavrilan, Felix Oberhansl, Alexander Wagner, Emanuele Strieder, and Andreas Zankl. Impeccable keccak towards fault resilient SPHINCS+ implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):154–189, 2024.

[HBG+18] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. Xmss: extended merkle signature scheme. RFC 8391, RFC Editor, May 2018.

[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.

[MCF19] D. McGrew, M. Curcio, and S. Fluhrer. Leighton-micali hash-based signatures. RFC 8554, RFC Editor, April 2019.

[Mer79] R. C. Merkle. *Secrecy, authentication, and public key systems.* Phd thesis, Stanford University Department of Electrical Engineering, 1979. [Dissertation].

[NIS] Post-quantum cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography. National Institute of Standards and Technology (U.S.). Accessed: 2025-08-27.

[NIS24] *Stateless Hash-Based Digital Signature Standard.* August 2024. National Institute of Standards and Technology (U.S.). NIST FIPS 205.

[Saa24] Markku-Juhani O. Saarinen. Accelerating SLH-DSA by two orders of magnitude with a single hash unit. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part I*, volume 14920 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 2024.