SHAKEing SLH-DSA with Non-profiled SCAs

Rafael Carrera Rodriguez[1], Jad Zahreddine[1,2], André Monthieu[1], Aurélien Vasselle[1], and Pierre-Yvan Liardet[1]

[1] eShard, France
[2] LIX, Inria, CNRS, École Polytechnique, Institut Polytechnique de Paris, France

Abstract. SLH-DSA is a hash-based digital signature algorithm, standardised by NIST as a post-quantum signature scheme. While prior work has analysed the vulnerability of its pseudorandom function (PRF) to differential side-channel analysis (SCA), no prior work has thoroughly examined this vulnerability under SHAKE as the PRF, nor recovered a full secret seed in practice. In this work, we address this gap by first analysing the differential SCA vulnerability of SLH-DSA-SHAKE, examining the inputs to the PRF and the possible selection functions for an attack. Then, based on this analysis, we propose a methodology to efficiently apply correlation power analysis (CPA) on every parameter set of SLH-DSA-SHAKE with three main steps: a strategy of averaging subtraces of the same signature to increase the signal-to-noise ratio; a partial-then-total CPA approach to reduce the complexity; and an efficient algorithm for the remaining exhaustive search to recover a full secret seed, which enables universal forgery. We then provide experimental results in simulation and on a 32-bit microcontroller. Our attack requires on average from 280 to 559 signatures, and in our experiments the exhaustive search completes within 45 minutes on 72 cores.

1 Introduction

From the seminal idea of Leslie Lamport in 1979 [Lam79] to build a signature scheme based on hash functions, many proposals have improved the approach practicality and efficiency. SLH-DSA [Nat24] is one of the most recent fruits of these works in the recent tree of hash-based signatures. In fact, the quantum threat on asymmetric cryptographic schemes, several new post-quantum-resistant ones have lately been standardized, and SLH-DSA is one of them. According to the standard, the possible hash families behind SLH-DSA may be either SHA-2 [Nat15a] or SHAKE (SHA-3) [Nat15b]. SPHINCS$^+$, on which SLH-DSA is based, is proven to be post-quantum existentially unforgeable under adaptive chosen-message attacks in the quantum-accessible random oracle model, under assumptions on its hash functions [BHK$^+$19a], with a formally verified proof [HK22,BDH$^+$24]. However, the vulnerability of SLH-DSA against side-channel analyses (SCA) remains to be studied [DRC$^+$25], with only a few works in the literature addressing this gap, including attacks and countermeasures [KGB$^+$18,BC23,Saa24,Flu24]. Specially, in SLH-DSA the static secret **SK**.seed is manipulated multiple times by the Pseudo-Random Function (PRF) alongside varying known data during signing, and its recovery would allow universal forgery of a signature. Such scenario may be vulnerable to non-profiled differential side-channel attacks as introduced by Kocher et al. [KJJ99]. While Kannwischer et al. [KGB$^+$18] showed how SPHINCS using a BLAKE-based PRF, the precursor of SLH-DSA, was vulnerable to differential SCA, the possibility of such analogue vulnerability in SLH-DSA using SHAKE is an open question. In this paper, we thus assess the vulnerability of SLH-DSA based on SHAKE against differential side-channel attacks. We track in the signature process how an attacker can derive DPA selection functions to take advantage of bit spreading and mixing during first layers of Keccak permutation. We derive CPA attacks and demonstrate them on a real device focussing on the use of Keccak in the complex context of SLH-DSA.

Contribution

Our contributions in this paper are described as follows:

- We provide an analysis of the variability of the inputs for the PRF of SLH-DSA, so that differential side-channel attacks can be optimized using this information.

– We propose a thorough side-channel attack procedure on the PRF of SLH-DSA, allowing partial recovery of the secret seed **SK**.seed using several CPAs, and completing the key recovery from this extracted information for all six SLH-DSA-SHAKE parameter sets. The remaining exhaustive search effort is reasonable, with complexity 2^6 to 2^{39}, depending on the scheme parametrization. The attack was successfully demonstrated on an STM32F446RE microcontroller, with an instrumented firmware based on the SPHINCS$^+$ reference implementation, requiring 280 to 559 signatures on average.
– We describe a trace averaging method for improving signal-to-noise ratio using traces from the same single signature.
– We provide an efficient verification method for a candidate for **SK**.seed using one valid signature. Instead of running the full expensive Key Generation, Sign or Verification, our oracle method requires one single hash function for comparison.

Related works

Kannwischer et al. [KGB$^+$18] first performed a DPA on the PRF of SPHINCS, albeit on a BLAKE-256-based implementation. They recover a chunk of the SPHINCS PRF seed using a six-stage DPA. Additionally, they targeted a SHA-2-based PRF for XMSS, finding that the input variability can hinder the success on an attack. The possibility of DPA leakage for the PRF is also extended to SLH-DSA both with SHAKE and SHA-256 PRFs, confirmed in practice by Saarinen [Saa24] via a TVLA on a naive, unaccelerated implementation.

Zohner et. al [ZKSH12] presented side-channel analysis of Keccak prior to its standardization by targeting the θ step. Taha and Schaumont [TS13a,TS13b] attacked Keccak with a DPA in the context of MAC-Keccak, targeting the leakage of θ , and of the χ step for long keys. In their work, they analyse the effect of the input format to Keccak on the success rate of the attack. Alternatively, Luo et al. [LFF$^+$15] target the output of the round permutation, profiting from the non-linearity of the χ step, while Chu and Lukowiak [CL18] pipeline attacks on both the θ step and the output of the round permutation.

Regarding profiled attacks on Keccak, You and Kuhn [YK20] attack an 8-bit implementation of SHA3-512, by templating on three intermediate states in the permutation function, and combining the likelihood of the possible values. Kannwischer et al. [KPP20] analyse the feasibility of soft-analytical side-channel attacks (SASCA) [VGS14] by simulating the leakage of different architecture sizes, obtaining good results on 8-bit architectures and moderate results on 32-bit ones. Finally, You and Kuhn [YK21] managed to apply SASCA for SHA3-512 on a 32-bit architecture, by creating fragment templates on parts of the full word.

Beyond side-channel analyses, a single faulty signature allows universal forgery on the SPHINCS framework [CMP18], as demonstrated with voltage glitches on a microcontroller [GKPM18] and adapted to SPHINCS$^+$ [Gen23]. Boy et al. [BPP$^+$25] recently achieved such a forgery on the SLH-DSA implementation of OpenSSL with Rowhammer.

While writing this related-work section, we found two publications [LCD$^+$26,CLD$^+$26], independent of our work, claiming a CPA attack on SLH-DSA. We analysed these papers and found several issues in them. We noted in these papers errors such as wrong selection functions, incomplete formulae and an imprecise leakage model for the attack, applied to a pure Python simulation. To the best of our knowledge, our aforementioned contributions remain novel in comparison to their work.

2 Background and Notation

2.1 SLH-DSA

SLH-DSA [Nat24] is a stateless hash-based digital signature scheme, derived from the SPHINCS$^+$ scheme [BHK$^+$19b]. It is composed of the following building blocks: a Few-Time-Signature scheme called Forest of Random Subsets (FORS); a One-Time-Signature scheme called Winternitz One-Time Signatures$^+$ (WOTS$^+$), and its use in the eXtended Merkle Signature Scheme (XMSS); and a hypertree of XMSS trees. The components are pipelined to build key generation, signing, and verification routines.

In a nutshell, SLH-DSA generates a signature of a message M as follows: it signs the digest of the message using a FORS instance, chosen from a function of the message. Then, a WOTS$^+$ instance signs the public key of such FORS instance. The public key of the WOTS$^+$ instance becomes a node of a Merkle tree (XMSS), where the nodes at the lowest level are WOTS$^+$ public keys. Upper layer nodes are a function of the lower

layer nodes. Eventually, the nodes converge into a single root node. An upper layer WOTS$^+$, member of another Merkle tree, signs the root node. The process repeats until calculation of the top node, which is the public key of the algorithm. The signing process is described in Figure 1, where h is the total height of the hypertree and d is the number of XMSS layers. The height of each XMSS tree is then $h' = h/d$. In Algorithm 1, M is the input of the internal functions of FIPS 205 [Nat24]; its encoding with a context string by the external interface does not affect our attack, and is therefore omitted.

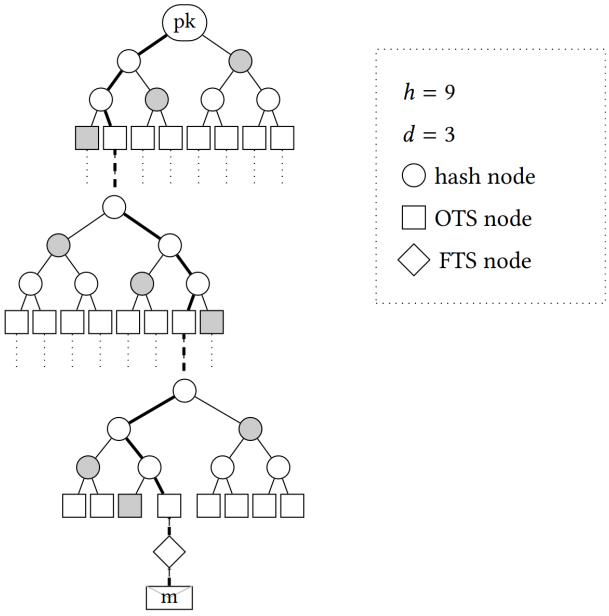


Fig. 1: Illustration of a simplified SLH-DSA structure, taken from [BHK$^+$19b].

The secret keys for the FORS and WOTS$^+$ instances in the tree are generated with a pseudo-random function (PRF) from a public key seed **PK**.seed, a secret key seed **SK**.seed and an address **ADRS**, defining the position in the tree and the type of algorithm for generation (see Section 3.4). The PRF can be instantiated with either SHA-256 or SHAKE256, depending on the chosen hash family for the entire algorithm: SHA-2 or SHAKE.

In addition to h, d, and the hash family, SLH-DSA is parameterized by the security parameter n (in bytes), the FORS tree height a and number of trees k, and the WOTS$^+$ Winternitz parameter lg_w. These in turn fix the message digest length m (in bytes). FIPS 205 approves twelve scheme parameter sets, listed in Table 1, pairing each of six (n, h, d, a, k, lg_w, m) combinations with either the SHA-2 or the SHAKE hash function family. The two hash function instantiations of a given size class share identical values for every parameter. The suffixes “s” and “f” denote the small-signature and fast-signing variants, respectively.

Table 1: SLH-DSA scheme parameter sets [Nat24], with the number of PRF calls with **SK**.seed in signature generation [Saa24], i.e. $k \cdot 2^a + d \cdot 2^{h'} \cdot len$ (one per FORS leaf and per WOTS$^+$ chain).

	n	h	d	a	k	lg_w	m	PRF calls
SLH-DSA-SHA2-128s **SLH-DSA-SHAKE-128s**	16	63	7	12	14	4	30	182,784
SLH-DSA-SHA2-128f **SLH-DSA-SHAKE-128f**	16	66	22	6	33	4	34	8,272
SLH-DSA-SHA2-192s **SLH-DSA-SHAKE-192s**	24	63	7	14	17	4	39	461,312
SLH-DSA-SHA2-192f **SLH-DSA-SHAKE-192f**	24	66	22	8	33	4	42	17,424
SLH-DSA-SHA2-256s **SLH-DSA-SHAKE-256s**	32	64	8	14	22	4	47	497,664
SLH-DSA-SHA2-256f **SLH-DSA-SHAKE-256f**	32	68	17	9	35	4	49	36,144

Algorithm 1: SLH-DSA.Sign_simplified(M, SK, *addrnd*)

Generate an SLH-DSA signature.
Input: Message M, private key SK = (**SK**.seed, **SK**.prf, **PK**.seed, **PK**.root), (optional) additional randomness *addrnd*.
Output: SLH-DSA signature SIG.

ADRS $\leftarrow$ toByte$(0, 32)$
opt_rand $\leftarrow$ *addrnd* ▷substitute *opt_rand* $\leftarrow$ PK.seed for the deterministic variant
$R \leftarrow \text{PRF}_{msg}(\mathbf{SK}.\text{prf}, \textit{opt_rand}, M)$ ▷generate randomizer
SIG $\leftarrow R$
digest $\leftarrow \text{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M)$ ▷compute message digest
$(md, idx_{tree}, idx_{leaf}) \leftarrow$ parseDigest(*digest*) ▷split *digest* into the message signed by FORS and the tree and leaf indices
ADRS.setTreeAddress(idx_{tree})
ADRS.setTypeAndClear(FORS_TREE)
ADRS.setKeyPairAddress(idx_{leaf})
$\text{SIG}_{FORS} \leftarrow$ fors_sign(md, **SK**.seed, **PK**.seed, **ADRS**) ▷$k \cdot 2^a$ PRF calls
SIG $\leftarrow$ SIG $\|$ SIG_{FORS}
$\text{PK}_{FORS} \leftarrow$ fors_pkFromSig(SIG_{FORS}, md, **PK**.seed, **ADRS**) ▷recover the FORS public key
$\text{SIG}_{HT} \leftarrow$ ht_sign(PK_{FORS}, **SK**.seed, **PK**.seed, idx_{tree}, idx_{leaf}) ▷climb the hypertree; $d \cdot 2^{h'} \cdot len$ PRF calls with SK.seed
SIG $\leftarrow$ SIG $\|$ SIG_{HT}
Return SIG

2.2 Keccak and SHAKE256

KECCAK is a family of functions used in the SHA-3 standard [Nat15b]. In the context of SLH-DSA, it is used for hashing and generating pseudo-random values. Since our point of interest is the PRF use case, we will focus on the details that are relevant to it.

The Keccak state. The KECCAK state is a b-bit string S. It can be mapped into a three-dimensional array defined as $A[x, y, z] = S[w(5y + x) + z]$. For the width $b = 1600$ used by every function defined in FIPS 202, the coordinates are in the ranges $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, where the lane size is $w = b/25 = 64$ [Nat15b]. FIPS 202 names the sub-arrays of A used below: a *lane* is the w-bit sub-array with fixed (x, y), a *row* the 5-bit sub-array with fixed (y, z), and a *column* the 5-bit sub-array with fixed (x, z).

We remark that the state can be naturally viewed as a 5×5 grid of 64-bit lanes, indexed by (x, y), which is useful for visualizing the state composition in our analyses. Additionally, for ease of notation in our formulas, we consider that the x and y coordinate operations are mod 5 and that the z coordinates are mod 64.

Byte and byte-string related notation. A byte string of size n is defined as $\mathbb{B}^n$. The term "lane" refers to a 64-bit view of a byte string. For a byte-string X, we denote $\lambda_X[i, z]$ for the bit z of the i-th 64-bit lane of X. We write $\lambda_X[i]$ for the full i-th lane when the bit position is not relevant. We denote $X[a : b]$ the slice of X from byte a (inclusive) to byte b (exclusive), as in FIPS 205 [Nat24]. For a byte x, we denote $x.\text{setBit}(i, v)$ the member function that sets the bit at position i of x to the value v.

The Keccak permutation function. This function applies 24 rounds of 5 subfunctions to the state, for each call to the function. We summarize the KECCAK permutation subfunctions with a high-level description and we refer the reader to FIPS 202 [Nat15b] for more details. The θ step calculates the bit parity of each column in the state. Then, each bit of the state is XORed with two specific column parities, one of them shifted by one, as shown in Equation (1). Two intermediate steps of θ can be considered in the column parity $C[x, z] = \oplus_{i=0}^{4} A[x, i, z]$ and the neighbouring column parity $D[x, z] = C[x-1, z] \oplus C[x+1, z-1]$. Therefore, $A_\theta[x, y, z] = A[x, y, z] \oplus D[x, z]$. The ρ and π steps rearrange the bits of the state. Afterwards come the χ step which XORs every bit in each row with the result of a non-linear function on two other bits in the row, as shown in Equation (2). Finally, during the ι step, a round constant is XORed to lane $(x, y) = (0, 0)$.

$$A_\theta[x, y, z] = A[x, y, z] \oplus (\oplus_{i=0}^{4} A[x-1, i, z]) \oplus (\oplus_{i=0}^{4} A[x+1, i, z-1]) \tag{1}$$

$$A_\chi[x, y, z] = A[x, y, z] \oplus ((A[(x+1), y, z] \oplus 1) \cdot A[(x+2), y, z]) \tag{2}$$

The Keccak sponge function. The sponge function *absorbs* the input by XORing it into the state r bits at a time, with a permutation after each block, then *squeezes* the output r bits at a time, with a permutation between output blocks. The rate r and the capacity c, the bits never involved in these two steps, satisfy $b = r + c$. The input is padded with pad10*1, which appends a 1, then the minimum number of zeros, and a final 1, so that the padded length is a multiple of r. KECCAK[c] denotes this sponge function with the KECCAK permutation of width $b = 1600$, capacity c and pad rule pad10*1.

SHAKE256 as SLH-DSA's PRF. SLH-DSA can instantiate its PRF with the extendable output function SHAKE256. SHAKE256 is instantiated with KECCAK[512] and the domain separation suffix 1111 such that $\text{SHAKE256}(M, \ell) = \text{KECCAK}[512](M||1111, \ell)$, where M is the input to hash and ℓ is the requested output length. Equation (3) shows the instantiation of the PRF with SHAKE256, having as inputs the known **PK**.seed, **ADRS** and the secret **SK**.seed.

$$\text{PRF}(\mathbf{PK}.\text{seed}, \mathbf{SK}.\text{seed}, \mathbf{ADRS}) = \text{SHAKE256}(\mathbf{PK}.\text{seed}||\mathbf{ADRS}||\mathbf{SK}.\text{seed}, 8n) \tag{3}$$

2.3 Correlation Power Analysis

The Correlation Power Analysis (CPA) [BCO04] is a side-channel attack that compares the leakage of an operation $\mathcal{O}$ with a leakage model $\mathcal{M}$ of such operation using Pearson's correlation coefficient as a statistical distinguisher. $\mathcal{O}$, which we call the *selection function*, must depend on a known variable input x and a static targeted secret s.

An attacker first obtains N side-channel traces of $\mathcal{O}$, one for each known input x_i, each with t time samples; $\boldsymbol{L}_j$ denotes the vector of the j-th time sample of every trace. For a guess s^* of s, the attacker computes the leakage model $m_i = \mathcal{M}(x_i, s^*)$ for each known input, usually the Hamming weight or the Hamming distance of the result of $\mathcal{O}$, and collects them in $\boldsymbol{M}$. For each time sample j, the attacker computes Pearson's correlation coefficient $r(\boldsymbol{L}_j, \boldsymbol{M})$, which gives a *correlation trace* of size t. This is repeated for every guess of s, and the guess with the highest absolute value in its correlation trace is chosen.

If s is too large, applying CPA to subparts s_i of s enables a divide-and-conquer strategy, which reduces the complexity of searching for s.

3 Exploring the Non-profiled Differential SCA vulnerability of SLH-DSA-SHAKE

The secret **SK**.seed, part of the full secret key **SK**, is used for deriving secret elements throughout the signing algorithm. In fact, the recovery of **SK**.seed alone suffices, together with the public key, to sign any message, allowing universal forgery. While **SK**.prf could be targeted by a similar attack on PRF_{msg}, its recovery is not needed. Let an adversary with knowledge of **SK**.seed choose any message M and an $R' \in \mathbb{B}^n$. This adversary runs a modified version of the Algorithm 1 to obtain a signature SIG, where the calculation of R in Line 3 is replaced by the chosen R'. This SIG will pass the verification, since a verifier cannot check if R' in SIG was generated honestly using Line 3 of Algorithm 1: in the verification of FIPS 205 [Nat24], R is only read from SIG and used as an input to H_{msg}, and recomputing it would require **SK**.prf.

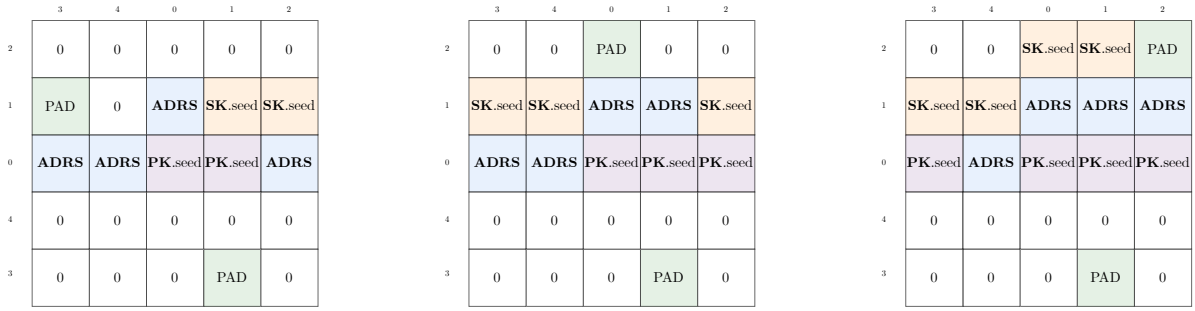
A differential side-channel attack can be mounted to recover **SK**.seed. The operation that deals with **SK**.seed is the PRF, which can be instantiated with either SHAKE256 or SHA-256. In this work, we focus on SHAKE256.

3.1 Possibility of horizontal attacks

As hinted by Saarinen [Saa24], the PRF being called multiple times per signature opens the door to so-called *horizontal attacks*, where the trace of a full signature is cut into subtraces of individual PRF calls. Additionally, if the same key pair is used for several signatures, the subtraces of multiple signatures can be combined to obtain more data. This is especially useful if the variability of **ADRS** is not sufficient to mount a DPA on the subtraces of a single signature. The last column of Table 1 gives the number of calls of the PRF per version of SLH-DSA during the signature generation. As expected, the "s" versions use considerably more PRF calls than the "f" versions.

3.2 Data absorbed in a Keccak state

After the absorb step in the KECCAK sponge function, the input data of the PRF fills the internal state. In Figure 2, we show the frontal view of the state for each security level of SLH-DSA-SHAKE. In these figures, the term PAD includes both the KECCAK padding function and the SHAKE256 suffix. We note that after the absorb, each 64-bit lane depends individually on one subpart of the inputs, and that both the "f" or "s" scheme parametrizations with the same security level yield the same state format.



(a) 128-bit (b) 192-bit (c) 256-bit

Fig. 2: KECCAK state frontal view for SLH-DSA-SHAKE scheme parametrizations. The `f` and `s` variants of each security level share the same state format.

3.3 Selection functions in the Keccak permutation

Attempting DPA requires a selection function that operates on both the targeted secret and known variable data. The internal subfunctions $(\theta, \rho, \pi, \chi, \iota)$ of the KECCAK permutation are such possible selection functions for targeting the recovery of **SK**.seed. The two main subfunctions that operate significantly on bits of the state are the θ and χ steps. The other steps only move bits (ρ and π) or add a constant (ι), so a leakage of their outputs reduces to a selection function on θ or χ, respectively. In the following, we therefore formulate the attacks in terms of θ, even though the actual leakage appears at the output of ρ.

We now focus on a possible attack on the first round of the KECCAK permutation. Targeting θ has the advantage of isolating the dependence of the selection function to a single bit (or lane) of **SK**.seed. For instance, in SLH-DSA-SHAKE-128, the application of θ to the lane $(x, y) = (2, 0)$ depends exclusively on $\lambda_{\mathbf{SK}.\text{seed}}[0]$, $\lambda_{\mathbf{ADRS}}[0]$, $\lambda_{\mathbf{ADRS}}[1]$, $\lambda_{\mathbf{PK}.\text{seed}}[1]$ and lanes of the padding. On the other hand, θ is based on XORs, which may increase the difficulty of side-channel analyses, due to its linearity.

The χ subfunction as a target provides the advantage of being a non-linear function. However, in its application in the first round, it involves at least two bits of **SK**.seed for each resulting bit. An attack on χ may require increased computational complexity compared to θ due to the increased set of considered secret hypotheses at a time.

Attacks on further rounds may be possible, but lack interest due to the diffusion property of the KECCAK permutation: every selection function will depend on multiple bits of the secret, which increases the computational complexity of the DPA.

Finally, since a single bit of **SK**.seed can appear in multiple selection functions, several leakage points are available for an attacker.

3.4 Analysis of the variability of known inputs

A DPA requires variable (and preferably uniformly distributed) inputs that are combined with the targeted secret. The known inputs of the PRF are **PK**.seed and **ADRS**, with **ADRS** being the only input that varies, albeit it cannot be directly controlled. As shown by Kannwischer et al. [KGB+18], a limited variability of inputs can hinder the execution of an attack against a PRF. Therefore, an analysis of the variability of **ADRS** can steer an attacker to choose a better suited selection function.

ADRS is a 32-byte string indicating the position in the hypertree of the value being computed. It is composed of 20 bytes for generic fields and 12 bytes for function-specific fields. The generic fields, in order, are the 4-byte layer address (the height of the tree in the hypertree, vertical and upwards), the 12-byte tree address (position of a tree in a layer, horizontal and left-to-right) and 4 bytes for the type, depending on the context where **ADRS** is being used. The layer address can take values from 0 to $d-1$, while the tree address in a layer L ranges from 0 to $2^{(d-1-L)h'} - 1$. Every address is encoded following the big-endian convention. Among the several types of **ADRS**, only two are used by the PRF, which define the format of the specific fields: the WOTS$^+$ key generation and the FORS key generation.

The two formats differ only in their type code and their three specific words; Figure 3 shows the resulting 32-byte field layout together with the value range of each field.

Variability of the ADRS fields. We now analyse in order each field of **ADRS** with respect to its variability, which itself is dependent on the scheme parametrization.

The layer address varies very little, taking at maximum the value of 21; for a 4-byte allocation of this address, this means that 27 bits are fixed at zero at all times.

The tree address has a larger and essentially uniform range for the layer 0; going up in the tree means dropping the h' least significant bits, therefore reducing the range as the path is climbed up. Here we remark a difference between the "s" and "f" variants: even though the "s" version is designed for smaller signatures and more calls to the PRF (an advantage for a side-channel attacker), such calls to the PRF are done over a more restricted input for the tree address, compared to the "f" variants. Therefore, attacks over the "f" versions may recover a larger part of **SK**.seed, even if the number of samples for the attacker is reduced, as we show in Section 4.3.

The type field only varies between 5 or 6, depending on whether the context is WOTS$^+$ or FORS. It may be of use for an attacker since the two least significant bits flip between the two different types. Nevertheless, the other 30 bits of the field are constant, 29 of them always fixed at zero.

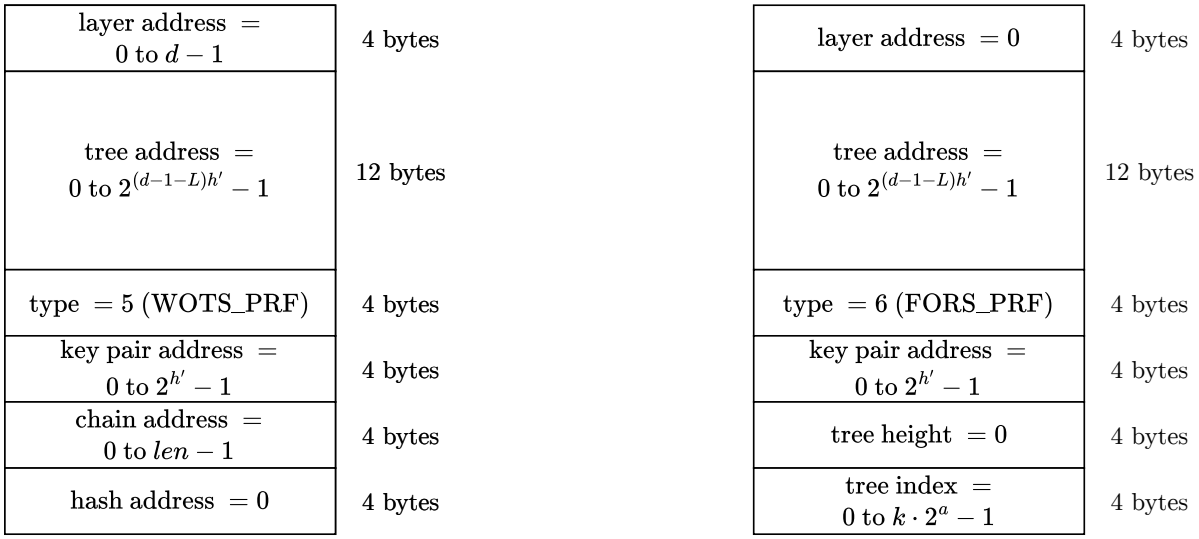


Fig. 3: Field layout and per-field value range of the 32-byte **ADRS** passed to the PRF, for (left) WOTS$^+$ key generation and (right) FORS key generation. $len = 2n + 3$.

Both WOTS$^+$ and FORS make use of the next specific field in the same way: a key pair address related to the leaf index of an XMSS tree. The value of this address is essentially uniform and it favours the attacks on the "s" versions, whose larger h' gives more varying bits. However, its range affects at most 9 bits of the field, leaving at least 23 bits always at zero.

Finally, for the last two fields, we distinguish the WOTS$^+$ and the FORS case. In the case of WOTS$^+$, the penultimate field for the chain address has a reduced but uniform range, since Algorithm 1 must iterate to get the secret value for each chain of WOTS$^+$; the last field, the hash address is always fixed at zero. On the other hand, for FORS, the penultimate field for the tree height is fixed at zero; the last field, the tree index is uniform for consecutive subranges of length 2^a, depending on the index of the set being generated (for the set i, it is uniform from $i \cdot 2^a$ to $(i+1) \cdot 2^a - 1$). Its range may span up to 19 bits, leaving at least 13 bits always at zero. One can imagine that an attacker that is interested in bits of **SK**.seed that interact with the penultimate field, may choose samples from WOTS$^+$, and samples from FORS otherwise if the interest is on the last field.

Considering the way data is absorbed in the Keccak state, shown in Figure 2, it may be interesting to focus on the fields of **ADRS** that fill a full 64-bit lane of the state, such that they are combined with as many bits of **SK**.seed as possible. The field that complies the best with this requirement is the generic tree address field, nearly or completely filling (depending on the scheme parametrization) the second lane that is allocated to **ADRS** in the state. An attacker may want to focus on selection functions involving this lane, which corresponds to the lanes $(x, y) = (3, 0)$, $(4, 0)$ and $(0, 1)$ for SLH-DSA-SHAKE-128, SLH-DSA-SHAKE-192 and SLH-DSA-SHAKE-256. With such a choice, the "f" versions provide better variability than the "s" versions.

Variability of ADRS per signature execution. Now we consider the possibility of horizontal attacks, by analysing the variability of **ADRS** layer by layer.

- **Layer 0:** In the bottommost layer of the signature execution, the PRF is used for both FORS and WOTS$^+$.
  - FORS: The PRF is called $k \times 2^a$ times. Every field is constant except for the last field, the FORS tree index. Every call of the PRF in this context, will take a single value in $[0, k \cdot 2^a - 1]$.
  - WOTS$^+$: The PRF is called for each WOTS$^+$ instance in the Merkle tree and their internal chains. That is, $2^{h'} \times len$ times. The fields that vary are the key pair address and the chain address; the former varying in the range of $[0, 2^{h'} - 1]$ and the latter in the range of $[0, len - 1]$.

– **Upper layers:** In the $d-1$ upper layers, the PRF is used exclusively for WOTS$^+$. For each layer, the variation is the same compared to first layer; the only difference is that as the signature path goes up the tree, the layer address is increased by one, while the tree address gets shifted h' bits to the right.

An attacker can analyse the PRF calls per layer for averaging purposes: if all the lanes of **ADRS** involved in a selection function do not change in a layer, the attacker may as well average the traces to augment the signal-to-noise ratio. **Mapping ADRS to lanes of a Keccak state. ADRS** maps to 4 lanes in the KECCAK state: $\lambda_{\mathbf{ADRS}}[i]$ for $i \in [0,3]$. Following our previous analysis, we show the contents of each lane along with the variability analysis.

– $\lambda_{\mathbf{ADRS}}[0]$: The 4-byte layer address field maps entirely to this lane along with 4 bytes always at zero from the tree address. It varies horizontally by incrementing its value by one for each layer.
– $\lambda_{\mathbf{ADRS}}[1]$: Contains only bytes from the tree address. Per signature, it has 63 varying bits for the scheme parametrizations "128f" and "192f", 64 for "256f", 54 for "128s" and "192s", and 56 for "256s". It also varies horizontally, albeit by shifting bits to the right, removing variance from the most significant bits.
– $\lambda_{\mathbf{ADRS}}[2]$: Contains 4 static bytes from the address type (either `FORS_PRF` or `WOTS_PRF`), and bytes from the key pair address (static for FORS, variable for WOTS$^+$). It varies per signature for FORS, and horizontally for WOTS$^+$. The number of varying bits is 3 for the "128f" and "192f" scheme parametrizations, 4 for "256f", 9 for "128s" and "192s", and 8 for "256s".
– $\lambda_{\mathbf{ADRS}}[3]$: Contains either the chain address and 4 bytes at zero for WOTS$^+$ or 4 bytes at zero and the tree index for FORS. It varies horizontally. For WOTS$^+$, the chain address provokes 6 varying bits for "128f", "192f", "128s" and "192s", and 7 for "256f" and "256s". For FORS, the tree index requires 12, 14 and 15 varying bits for "128f", "192f" and "256f" respectively, and 16, 19 and 19 for "128s", "192s" and "256s" respectively.

The lane that varies the most per signature call is then $\lambda_{\mathbf{ADRS}}[1]$. In the following section we describe our CPA attack focused on such lane.

4 CPA Attack on SLH-DSA

Considering the vulnerabilities shown in Section 3, we propose a CPA attack for every scheme parametrization of SLH-DSA. Figure 4 shows the graphical representation of the overall attack workflow and the following section describes in details such attacks.

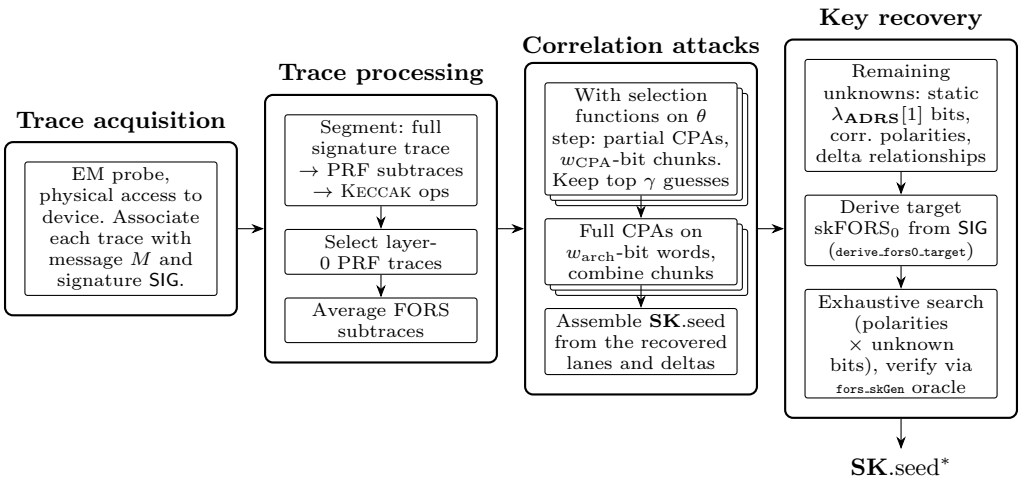


Fig. 4: Flow of the attack procedure.

4.1 Attacker model

The attacker is a passive observer that eavesdrops on the SLH-DSA protocol. They have physical access to the device, such that they can measure cryptographic activity through a side-channel. They know the public key used, and capture pairs of input messages and output signatures, along with the corresponding side-channel traces. In our case, an electromagnetic probe captures radiation during the signatures.

From these public data, and R taken from the signature, the attacker has to recompute $digest \leftarrow \mathrm{H}_{msg}(R, \mathbf{PK}.\mathrm{seed}, \mathbf{PK}.\mathrm{root}, M)$, and derive the tree and leaf indices, hence the **ADRS** of every PRF call (Algorithm 2). Here, M denotes the message as input to the internal signing function of FIPS 205; with the external interface, M is replaced by its encoding with the context string [Nat24], which is also public to any verifier.

Moreover, the attacker is assumed to know the position and order of the PRF calls within a signature. With this knowledge, the attacker can perform horizontal trace segmentation, extracting PRF subtraces from a full SLH-DSA execution, and then locate individual operations within the KECCAK permutation of each individual PRF execution. In our practical demonstration (Section 5), the segmentation step is not required, since custom firmware with triggers is used for convenience. Locating the KECCAK patterns within a full signature trace is a device-specific signal-processing step, which we do not detail, as the core of our contribution lies in the side-channel exploitation of the PRF and the recovery of **SK**.seed.

4.2 Trace selection and averaging

Following the analysis of the variability of **ADRS** in Section 3.4, we focus on maximizing the exposure to $\lambda_{\mathbf{ADRS}}[1]$. We propose selecting exclusively the side-channel traces from the execution of the PRF in layer 0, since this layer maximizes the number of variable bits. Additionally, we remark that in the PRF calls for FORS, every lane is constant, except for $\lambda_{\mathbf{ADRS}}[3]$. If the selection function for the CPA does not include such lane, it is entirely possible to average all the side-channel traces from FORS to maximize the signal-to-noise ratio. In this way, we take advantage of the multiple PRF calls in a single signature in the context of a multi-signature attack.

4.3 Target Intermediate Values

We select operations from the θ step as selection functions, since its easiness to isolate the leakage to a single bit (resp. lane) of **SK**.seed. Additionally, we focus on the operations that involve $\lambda_{\mathbf{ADRS}}[1]$ and that exclude $\lambda_{\mathbf{ADRS}}[3]$, so that trace averaging is possible. Finally, we simplify the selection functions for when $\lambda_{\mathrm{PAD}}[i]$ is entirely at zero.

We note that excluding operations that operate on $\lambda_{\mathbf{ADRS}}[3]$, excludes the usage of the χ step. In fact, the χ step always operates on $\lambda_{\mathbf{ADRS}}[3]$, for every scheme parametrization, since θ spreads it over two columns of the state and π distributes them across all rows.

SLH-DSA-SHAKE-128: The considered selection functions for these scheme parameters are at the output of θ. For $\lambda_{\mathbf{SK}.\mathrm{seed}}[0, z]$, it corresponds to the position $(x, y) = (2, 0)$ of the KECCAK state array, shown in Equation (4); for $\lambda_{\mathbf{SK}.\mathrm{seed}}[1, z]$, it corresponds to the position $(x, y) = (3, 0)$, shown in Equation (5).

$$\begin{aligned} A_\theta[2,0,z] = \lambda_{\mathbf{ADRS}}[1, z-1] \oplus \lambda_{\mathbf{ADRS}}[0, z] \oplus \lambda_{\mathrm{PAD}}[8, z] \\ \oplus \lambda_{\mathrm{PAD}}[0, z-1] \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[1, z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[0, z] \end{aligned} \tag{4}$$

$$A_\theta[3,0,z] = \lambda_{\mathbf{ADRS}}[2, z-1] \oplus \lambda_{\mathbf{ADRS}}[1, z] \oplus \lambda_{\mathbf{ADRS}}[0, z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[1, z] \tag{5}$$

SLH-DSA-SHAKE-192: $\lambda_{\mathbf{SK}.\mathrm{seed}}[2, z]$ is first targeted using the parity calculation in θ (Equation (1)) with $x = 4$, shown in Equation (6). For $\lambda_{\mathbf{SK}.\mathrm{seed}}[1, z]$ the selection function follows the output of θ at $(x, y) = (4, 0)$, as shown in Equation (7). The lane $\lambda_{\mathbf{SK}.\mathrm{seed}}[0]$ is not the only secret lane combined with $\lambda_{\mathbf{ADRS}}[1, z]$. It requires targeting a delta $\Delta\mathbf{SK}.\mathrm{seed}_{192,0,2} = \lambda_{\mathbf{SK}.\mathrm{seed}}[0, z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2, z-1]$, as shown in Equation (8). Each CPA targets the XOR of all the coloured secret terms of its selection function as a single unknown, so that the CPAs are independent of each other. The colours only indicate how the recovered

deltas are solved in the final key recovery step: the lane in red is obtained from the delta once the lanes in orange are known.

$$C[4,z] = \lambda_{\mathbf{ADRS}}[1,z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2,z] \tag{6}$$

$$\begin{aligned} A_\theta[4,0,z] = \lambda_{\mathbf{ADRS}}[2,z-1] \oplus \lambda_{\mathbf{ADRS}}[1,z] \oplus \lambda_{\mathbf{ADRS}}[0,z] \\ \oplus \lambda_{\mathrm{PAD}}[0,z-1] \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[0,z-1] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[1,z] \end{aligned} \tag{7}$$

$$\begin{aligned} A_\theta[3,0,z] = \lambda_{\mathbf{ADRS}}[1,z-1] \oplus \lambda_{\mathbf{ADRS}}[0,z] \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[2,z] \\ \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2,z-1] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[0,z] \end{aligned} \tag{8}$$

SLH-DSA-SHAKE-256: The selection function for $\lambda_{\mathbf{SK}.\mathrm{seed}}[2,z]$ is the parity calculation in θ (Equation (1)) at $x = 0$, in Equation (9). All the other lanes must be obtained via deltas. $\lambda_{\mathbf{SK}.\mathrm{seed}}[0,z]$ is targeted with the output of θ at $(x,y) = (4,0)$, using the relationship $\Delta\mathbf{SK}.\mathrm{seed}_{256,0,2} = \lambda_{\mathbf{SK}.\mathrm{seed}}[0,z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2,z-1]$, shown in Equation (10). $\lambda_{\mathbf{SK}.\mathrm{seed}}[1,z]$ is obtained with the output of θ at $(x,y) = (4,1)$, with the delta $\Delta\mathbf{SK}.\mathrm{seed}_{256,0,1,2} = \lambda_{\mathbf{SK}.\mathrm{seed}}[0,z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[1,z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2,z-1]$, in Equation (11). Finally, $\lambda_{\mathbf{SK}.\mathrm{seed}}[3,z]$ is obtained with the output of θ at $(x,y) = (0,1)$ and the relationship $\Delta\mathbf{SK}.\mathrm{seed}_{256,1,3} = \lambda_{\mathbf{SK}.\mathrm{seed}}[1,z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[3,z-1]$, as shown in Equation (12).

$$C[0,z] = \lambda_{\mathbf{ADRS}}[1,z] \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[0,z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2,z] \tag{9}$$

$$\begin{aligned} A_\theta[4,0,z] = \lambda_{\mathbf{ADRS}}[1,z-1] \oplus \lambda_{\mathbf{ADRS}}[0,z] \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[3,z] \\ \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[0,z-1] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2,z-1] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[0,z] \end{aligned} \tag{10}$$

$$\begin{aligned} A_\theta[4,1,z] = \lambda_{\mathbf{ADRS}}[1,z-1] \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[3,z] \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[0,z-1] \\ \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[2,z-1] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[1,z] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[0,z] \end{aligned} \tag{11}$$

$$\begin{aligned} A_\theta[0,1,z] = \lambda_{\mathbf{ADRS}}[2,z-1] \oplus \lambda_{\mathbf{ADRS}}[1,z] \oplus \lambda_{\mathbf{ADRS}}[0,z] \oplus \lambda_{\mathrm{PAD}}[4,z-1] \\ \oplus \lambda_{\mathbf{PK}.\mathrm{seed}}[1,z-1] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[3,z-1] \oplus \lambda_{\mathbf{SK}.\mathrm{seed}}[1,z] \end{aligned} \tag{12}$$

Limited ADRS variability and Remaining entropy Depending on the scheme parametrization, the variable ADRS actually contains some static bits in lane $\lambda_{\mathbf{ADRS}}[1,z]$. This is due to the range of the tree address in $[0, 2^{(d-1-L)h'} - 1]$, which may or may not cover the entirety of the 64 bits of the lane of $\lambda_{\mathbf{ADRS}}[1,z]$. For our attack we consider only the layer $L = 0$ and map the static bits to the lane for each scheme parametrization. For "128f" and "192f", only one bit is static in $\lambda_{\mathbf{ADRS}}[1,z]$, while every bit varies in "256f". The "s" versions have less variability: "128s" and "192s" have 10 static bits, while the "256s" version has 8 static bits. However, $\lambda_{\mathbf{ADRS}}[2]$ varies more for the "s" versions and yields one extra bit ($z = 0$) when it is involved (Equations (5), (7) and (12)), which we mark with $*$ in Table 2. For "128s" and "192s" this directly recovers one more bit of **SK**.seed, lowering the unrecoverable count by one. For "256s", the $\lambda_{\mathbf{ADRS}}[2]$ information instead takes the form of a *relationship* between two otherwise-unrecoverable bits of **SK**.seed—an equality between a bit of lane 1 and a bit of lane 3 (bits 72 and 199)—so that one bit is determined by the other and the number of bits left to search still drops by one. Table 2 summarizes the static bits of $\lambda_{\mathbf{ADRS}}[1]$ for each version, along with their implications for the recovery of **SK**.seed and the remaining entropy after the CPA step.

Table 2: Static bits of $\lambda_{\mathbf{ADRS}}[1]$ for each SLH-DSA scheme parametrization, along with their consequences on unrecoverable bits and remaining entropy. $*$ marks an extra bit obtained via $\lambda_{\mathbf{ADRS}}[2]$: recovered directly for "128s" and "192s", or implied by a bit relationship for "256s".

Version	Static bits of $\lambda_{\mathbf{ADRS}}[1, z]$	Unrecoverable bits	Remaining entropy (bits)
128f	$z = 7$	$\lambda_{\mathbf{SK}.\text{seed}}[0, z]$: $z = 8$ $\lambda_{\mathbf{SK}.\text{seed}}[1, z]$: $z = 7$	2
128s	$z \in [0:8] \cup [14:16]$	$\lambda_{\mathbf{SK}.\text{seed}}[0, z]$: $z \in [1:9] \cup [15:17]$ $\lambda_{\mathbf{SK}.\text{seed}}[1, z]$: $z \in [1:8] \cup [14:16]$	19*
192f	$z = 7$	$\lambda_{\mathbf{SK}.\text{seed}}[2, z]$: $z = 7$ $\lambda_{\mathbf{SK}.\text{seed}}[1, z]$: $z = 7$ $\Delta\mathbf{SK}.\text{seed}_{192,0,2}$: $z = 8$	3
192s	$z \in [0:8] \cup [14:16]$	$\lambda_{\mathbf{SK}.\text{seed}}[2, z]$: $z \in [0:8] \cup [14:16]$ $\lambda_{\mathbf{SK}.\text{seed}}[1, z]$: $z \in [1:8] \cup [14:16]$ $\Delta\mathbf{SK}.\text{seed}_{192,0,2}$: $z \in [1:9] \cup [15:17]$	29*
256f	None	None	0
256s	$z \in [0:8]$	$\lambda_{\mathbf{SK}.\text{seed}}[2, z]$: $z \in [0:8]$ $\Delta\mathbf{SK}.\text{seed}_{256,0,2}$: $z \in [1:9]$ $\Delta\mathbf{SK}.\text{seed}_{256,0,1,2}$: $z \in [1:9]$ $\Delta\mathbf{SK}.\text{seed}_{256,1,3}$: $z \in [1:8]$	31*

4.4 Balancing complexity and resilience for the CPA

Moreover, as we target 32-bit XOR operations, each guess and its bit-negation have by construction the same absolute correlation, and we have to account for these additional possibilities. This leaves one unknown correlation polarity bit per attacked 32-bit word, to be recovered by the exhaustive search (Section 4.5).

Performing the CPA on the full word size w_{arch} of an architecture may be prohibitive in complexity. Instead, we divide the attack on smaller chunks with word size $w_{\text{CPA}} = 8$, and consider all the other bits as algorithmic noise. A partial correlation still exists, with the downside of having an attenuation factor of $\sqrt{\frac{w_{\text{CPA}}}{w_{\text{arch}}}}$ [BCO04,THM+07]. For $w_{\text{arch}} = 32$, the attenuation factor corresponds to 0.5.

The operations in the θ step are XORs. A CPA on an XOR produces pairs of complementary guesses (differing with a Hamming distance of w_{CPA}) that share the same correlation coefficient in absolute value but with opposite sign.

For the first partial CPA we proceed as follows: We identify a time frame characteristic of the operation. We first characterized the timing of the leakages within the Keccak permutation, which gives, for each w_{arch}-bit store, a frame of 10 samples around its leakage; the correlation of all its subwords is then taken as the maximum over this frame. Since this partial correlation remains vulnerable to algorithmic noise, we retain the best γ guesses with positive correlation [TS13b], which are then extended to the full w_{arch}-bit word, following a two-stage extend-and-prune strategy [THM+07]. The rationale is that, even when the correct value is rather the complement, every subword should have a correlation of the same sign. Otherwise, in the full w_{arch}-bit word, the correlation values would cancel each other.

A final CPA is done on a full word size w_{arch} using the previously obtained γ guesses per subword, testing all possible combinations. The number of guesses in this step will be then $\gamma^{w_{\text{arch}}/w_{\text{CPA}}}$. After this step, the guess (and its complementary couple) that maximizes the value of the correlation coefficient is taken. Given the number of w_{arch}-bit words and the complementary guesses, the remaining entropy after the CPA steps is $\frac{n \cdot 8}{w_{\text{arch}}}$ bits.

4.5 Verifying a seed candidate and the remaining exhaustive search

After a successful CPA step, the recovered secret seed candidate $\mathbf{SK}$.seed still carries three kinds of residual uncertainty that the exhaustive search must resolve:

1. the *unrecoverable bits*: the bits of $\mathbf{SK}$.seed whose associated column of $\lambda_{\mathbf{ADRS}}[1]$ shows no variance over the traces and can therefore not be recovered by the CPA. Their positions are collected in the vector P (see Table 2);

2. the *polarity guesses*: the CPA on the XOR of the θ step recovers each w_{arch}-bit word of the *recovered value* only up to its complement, i.e. up to its *polarity*. As each lane of **SK**.seed is an XOR-linear transform (the ρ rotation and, where applicable, the cross-lane deltas) of those recovered values, this leaves $W = \frac{n \cdot 8}{w_{\text{arch}}}$ undetermined polarities;
3. the *relationships*: pairs of unrecoverable bits that a recovered bit of a cross-lane delta ties together, either as an equality or as a complement. Each such relationship removes one degree of freedom; they are collected in the set $\mathcal{R}$, with $r = |\mathcal{R}|$. Across our six attacks only `256s` exhibits one: the equality between bits 72 and 199 of **SK**.seed noted above.

Relationships, when present, tie one unrecoverable bit to another and are folded into the search rather than enumerated, leaving $u = |P| - |\mathcal{R}|$ *free* unrecoverable bits (the remaining-entropy count of Table 2). The total remaining entropy is thus $u + W$ bits; its value for each scheme parametrization is reported in Table 5.

The key candidate verification oracle is a stripped-down version of the SLH-DSA.Sign algorithm: from the signature we fetch the randomness R and the first FORS secret key skFORS_0, we recompute the digest and the address **ADRS** of skFORS_0, and we recompute the first FORS secret key from a seed candidate with a single PRF call through fors_skGen. A candidate is correct when its recomputed value matches skFORS_0. As the search space is far smaller than 2^{8n}, hash collisions are not a concern.

We split this into two parts. Algorithm 2 derives, once, the verification target skFORS_0 together with the public inputs needed to recompute it (**ADRS** and the FORS index *indices*[0]); it costs one hash operation (H_{msg}), or none if the digest is cached. Algorithm 3 then performs the exhaustive search, recomputing a candidate's skFORS_0^* with fors_skGen and comparing it to the fixed target, so that the per-candidate cost reduces to a single PRF evaluation and one comparison.

Algorithm 2: derive_fors0_target(M, SIG, PK)

Builds the verification target skFORS_0 *and the inputs needed to recompute it. We use the helper functions of FIPS 205 [Nat24].*

Input: Message M, Signature SIG, public key PK = (**PK**.seed, **PK**.root).
Output: Target skFORS_0, address skADRS of the first FORS secret key.

$\text{SIG}_{FORS} \leftarrow \text{SIG}.\text{getSIG_FORS}()$
$\text{skFORS}_0 \leftarrow \text{SIG}_{FORS}.\text{getSK}(0)$ ▷`first secret key revealed in the FORS signature`
$R \leftarrow \text{SIG}.\text{getR}()$
$digest \leftarrow \text{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M)$ ▷`compute message digest`
$md \leftarrow digest[0 : \lceil k \cdot a/8 \rceil]$
$tmp_idx_{tree} \leftarrow digest[\lceil k \cdot a/8 \rceil : \lceil k \cdot a/8 \rceil + \lceil (h - h/d)/8 \rceil]$
$tmp_idx_{leaf} \leftarrow digest[\lceil k \cdot a/8 \rceil + \lceil (h - h/d)/8 \rceil : \lceil k \cdot a/8 \rceil + \lceil (h - h/d)/8 \rceil + \lceil h/(8 \cdot d) \rceil]$
$idx_{tree} \leftarrow \text{toInt}(tmp_idx_{tree}, \lceil (h - h/d)/8 \rceil) \bmod 2^{h-h/d}$
$idx_{leaf} \leftarrow \text{toInt}(tmp_idx_{leaf}, \lceil h/(8 \cdot d) \rceil) \bmod 2^{h/d}$
ADRS $\leftarrow \text{toByte}(0, 32)$
ADRS.setTreeAddress(idx_{tree})
ADRS.setKeyPairAddress(idx_{leaf})
$indices \leftarrow \text{base_}2^b(md, a, k)$
skADRS $\leftarrow$ **ADRS** ▷`fors_skGen internal code`
skADRS.setTypeAndClear(FORS_PRF)
skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
skADRS.setTreeIndex(indices[0])
Return (skFORS_0, skADRS)

Algorithm 3 details the exhaustive search. A polarity flip complements a w_{arch}-bit word of a *recovered value*, not of the assembled seed; since each lane of **SK**.seed is an XOR-linear transform of those values, it maps to a *flip group* of seed bits that is in general neither word-aligned nor confined to a single lane. We therefore precompute the W polarity masks $\mathcal{M}_w = L(e_w)$ by propagating each recovered-word complement e_w through the XOR-linear reconstruction L of **SK**.seed. Candidates are then generated on the fly with negligible memory: an outer loop enumerates the 2^W polarity combinations (XORing the selected masks)

and an inner loop the $2^{|F|}$ assignments of the free unknown bits F (the positions P minus the dependent bit of each relationship), each relationship being enforced by tying one bit of its pair to the other. A candidate is validated by a single fors_skGen call against the target of Algorithm 2; the search is parallelized over the cores of one machine and returns on the first match. In the implementation, relationships are folded into the masks so they add no per-candidate cost, and a prior polarity characterization lets the attacker skip the outer loop.

Algorithm 3: exhaustive_search(**SK**.seed, skFORS$_0$, **PK**.seed, **ADRS**, $\mathcal{M}$, P, $\mathcal{R}$)

*Resolves the residual uncertainty of **SK**.seed by exhaustive search.*

Input: SCA-recovered seed **SK**.seed $\in \mathbb{B}^n$ (assembled from the CPA guesses),
target skFORS$_0$, public key **PK**.seed, FORS address **ADRS**,
polarity masks $\mathcal{M} = (\mathcal{M}_0, \ldots, \mathcal{M}_{W-1})$ with $W = n \cdot 8/w_{\text{arch}}$,
unrecoverable-bit positions P, relationships $\mathcal{R} \subseteq P \times P \times \{0,1\}$.

Output: Candidate secret key seed **SK**.seed*, or $\perp$.

$F \leftarrow P \setminus \{\, b : (a,b,c) \in \mathcal{R} \,\}$ ▷`free bits: dependent bits removed,` $|F| = |P| - |\mathcal{R}|$
for i **from** 0 **to** $2^W - 1$ **do** ▷i `enumerates the polarity combinations`
  SK.seed$^* \leftarrow$ **SK**.seed
  for w **from** 0 **to** $W - 1$ **do**
    if $(i >> w) \bmod 2 == 1$ **then SK**.seed$^* \leftarrow$ **SK**.seed$^* \oplus \mathcal{M}_w$; ▷`apply the` w`-th polarity flip`
  for j **from** 0 **to** $2^{|F|} - 1$ **do** ▷j `enumerates the free unknown-bit assignments`
    for m **from** 0 **to** $|F| - 1$ **do SK**.seed$^*[\lfloor F[m]/8 \rfloor]$.setBit($F[m] \bmod 8$, $(j >> m) \bmod 2$);
    foreach $(a,b,c) \in \mathcal{R}$ **do**
      $v_a \leftarrow$ **SK**.seed$^*[\lfloor a/8 \rfloor]$.getBit($a \bmod 8$)
      SK.seed$^*[\lfloor b/8 \rfloor]$.setBit($b \bmod 8$, $v_a \oplus c$) ▷`enforce the relationship`
    skFORS$_0^* \leftarrow$ SHAKE256(**PK**.seed$\|$**ADRS**$\|$**SK**.seed$^*, 8n$) ▷`PRF expression`
    if skFORS$_0^*$ == skFORS$_0$ **then Return SK**.seed*;
Return $\perp$

5 Experimental Results

5.1 Code analysis

Since our target is only the PRF of SLH-DSA, for ease of trace capturing, we built a firmware image with only the PRF receiving **ADRS**, **SK**.seed and **PK**.seed as arguments. Restricting the acquisition to the PRF executions also keeps the dataset, already several gigabytes, to a size that can be shared as an artifact. The generation of the inputs (**ADRS** of each signature) is provided with the artifact. The PRF code itself is taken from the SPHINCS$^+$ submission repository in branch `consistent-basew` at commit `129b72c`[3]. Specifically, the only two files that were taken are `ref/fips202.c` and `ref/hash_shake.c` (and their headers). For `ref/hash_shake.c`, the only function that was kept was `prf_addr`, since any other function was not relevant for the PRF. For `ref/fips202.c`, two changes were made. First, since we are only interested in the first round of the Keccak permutation, the function `KeccakF1600_StatePermute` was modified so that an arbitrary number of rounds can be executed. Second, a GPIO trigger was added around the loop for the Keccak rounds, for ease of identification in a side-channel trace. A firmware image with communication utilities was compiled for each scheme parametrization, targeting the ARM Cortex M4-based STM32F446RE microcontroller, using the `-Os` optimization target. We observed that the compiled code for the Keccak permutation is identical for all the scheme parametrizations, except for the memory location for the round constants, which is not relevant for our analysis. Therefore, any code analysis applies to all the scheme parametrizations.

We target the `STR` instructions that write the attacked values to memory, since each triggers a write operation on the SRAM bus, which, from experience, leaks significantly more than the CPU logic on the

[3] https://github.com/sphincs/sphincsplus/tree/129b72c80e122a22a61f71b5d2b042770890ccee

STM32F4. We model this leakage with the Hamming weight of the stored value. Therefore, our goal is to bind the selection functions shown in Section 4.3 to a `STR` instruction. Since the word size is $w_{\text{arch}} = 32$, every 64-bit Keccak operation is performed in two steps: one for the higher half-lane and another one for the lower half-lane. For the parity calculations $C[x, z]$ in θ, the firmware does not store values in memory unless they are shifted by one during the calculation of $D[x, z]$ (except for the parity calculation of $x \in [3, 4]$ which is stored directly). For the outputs of θ, they are never directly stored in memory: the shifting ρ step is performed immediately and only after, the results are stored. As an example, **??** 1.1 shows the instructions for the θ and ρ steps for position $(x, y) = (2, 0)$, with a corresponding ρ offset of 62. The finalization of the θ step, and the ρ step are consecutive, followed by `STR` instructions.

```
...
0x08001A7C  EOR.W   R3, R3, R8        ; R3, R8 -> lower half-lanes of A[2,0,z] and D[2, z]
0x08001A80  EOR.W   R4, R4, R9        ; R4, R9 -> higher half-lanes of A[2,0,z] and D[2, z]
0x08001A84  MOV     R8, R3
0x08001A86  LSRS    R3, R3, #2        ; rho shift of 62 (2 to the right) lower half-lane
0x08001A88  ORR.W   R3, R3, R4,LSL#30 ; consider spilling bits from higher half-lane
0x08001A8C  STR     R3, [SP,#0x180]   ; store lower half-lane
0x08001A8E  LSRS    R3, R4, #2        ; rho shift of 62 (2 to the right) of the higher part
0x08001A90  ORR.W   R3, R3, R8,LSL#30 ; consider spilling bits from the lower part
...
0x08001A98  STR     R3, [SP,#0x184]   ; store higher half-lane
...
```

Listing 1.1: Extract of the compiled `-Os` firmware where the end of θ and the ρ step are consecutive, for $(x, y) = (2, 0)$. The result is written to memory with two `STR` instructions.

Following these observations, we should ensure our final selection functions are shifted according to the values actually stored: when they involve the parity in θ, we use their shifted-by-one version; when they involve the output of θ, we consider the shift given by ρ. The resulting selection functions, with their ρ offsets, are given in full for each scheme parametrization in the artifact.

5.2 Simulation campaign

In order to quickly validate the methodology, we first ran the attacks on emulated traces using our in-house QEMU-based tool. For each traced assembly instruction of the firmware, a sample is computed as the sum of the Hamming Weight of the registers that are read and/or written. Three noise-free datasets of the fast SLH-DSA versions were then generated, with the tracing limited to the first two rounds of the Keccak-f permutation.

To justify the extend-and-prune strategy (8-bit-then-w_{arch}-bit, Section 4.4) over the plain 8-bit one, we compared the minimum amount of traces required for the CPA to succeed for different Gaussian noise levels. The parameter γ was set to 9 as XOR leakages tend to correlate highly with the true secret, then with the group of 8 guesses at Hamming Distance 1. This results in the complexity of the w_{arch}-bit attack to be $9^4 < 2^{13}$. Table 3 shows that the extend-and-prune attack requires fewer traces up to $\sigma = 8$, with the largest reduction at $\sigma = 2$, and brings no gain from $\sigma = 9$, which is expected as it only mitigates the algorithmic noise.

Table 3: Average number of required traces, over 20 successful trials, of the plain 8-bit attack against the extend-and-prune attack ($\gamma = 9$) for different noise levels. σ corresponds to the standard deviation of the added Gaussian noise centered around zero.

Noise σ	0	1	2	3	4	5	6	7	8	9	10
8-bit attack #traces	80	106	172	248	327	429	668	793	1053	1263	1503
Extend-and-prune #traces	60	66	85	147	253	397	557	719	1010	1311	1583
Reduction %	25	37.7	50.4	40.9	22.8	7.5	16.6	9.3	4.1	-3.8	-5.3

5.3 Real device campaign

Bench setup The experiment was run on a NUCLEO STM32F446RE with a 30 MHz CPU clock. The traces were acquired with a Langer EMV LF-B 3 electromagnetic probe, a Langer EMV PA 303 SMA 30 dB amplifier and a LeCroy HDO6104B-MS oscilloscope. The sampling rate was set to 250 MS/s and a GPIO was used to frame the operation of interest. In order to perform a cartography of the leaking spots on the area of the chip, the board was placed on a Thorlabs MCM301 motorized stage. The metric for the cartography was the correlation of the trace with the first 32 output bits of the first two rounds of Keccak. The position chosen for the EM probe was the bottom right of the chip. The bench and the result of the cartography are shown in Figure 5.

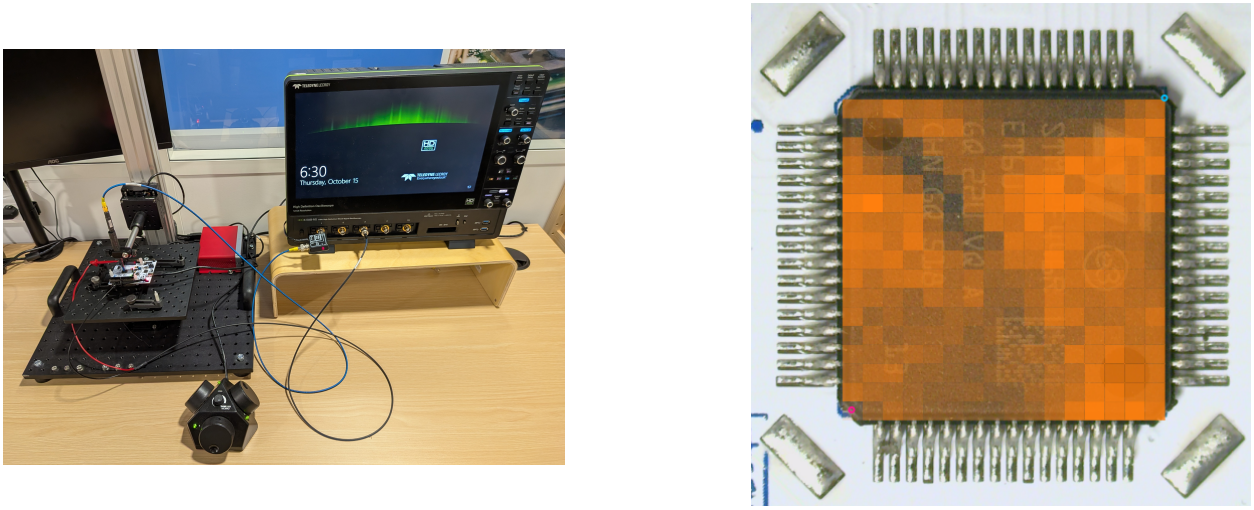


Fig. 5: Hardware bench setup of the real device campaign (left), and EM cartography results using as metric the trace correlation with the first 32 output bits of the first two rounds of Keccak (right).

Results As mentioned, the noise was mitigated by averaging the PRF calls related to FORS, which have identical first three lanes of the address. For trace capturing speed purposes, we chose an averaging factor of 100 traces per signature, thus discarding most of the PRF invocations.

According to Table 4, the full set of CPAs requires on average fewer than 600 signatures for any of the six parametrizations of SLH-DSA. For comparison, Kannwischer et al. [KGB+18] used 10,000 traces, corresponding to about 2,000 signatures, for the first two stages of their DPA on SPHINCS. With our strategy, the side-channel analysis does not show any significant difference in attacking one parametrization or the other, except for the number of secret lanes to recover, which dictates the total number of CPAs. The major difference in the complexity of the attack thus comes from the exhaustive search. Even if the SCA succeeds with no guessing entropy left, the exhaustive key search (Section 5.4) still has to cover a space of size defined by the product of the unrecoverable bits and the $\frac{n}{4}$ unknown correlation polarity bits. In particular, the "f" versions do not require fewer traces than the "s" versions, despite the better variability of their tree address (Section 3.4); their advantage appears instead in the exhaustive search, with fewer unrecoverable bits (Table 5).

5.4 Exhaustive search results

We ran Algorithm 3 on the seed recovered by the CPAs for each scheme parametrization, validating candidates against the FORS target derived with Algorithm 2 from the public signature. This completes the six attack procedures, one per SLH-DSA-SHAKE parameter set, each on its own dataset.

Table 4: Average, lowest and greatest number of required signatures over 8 datasets for the set of $\frac{n}{4}$ CPAs to succeed per version with $\gamma = 9$. Each signature provides 100 PRF subtraces averaged together, and all 8 datasets come the same acquisition campaign with the same **SK**.seed.

Version	Average	Lowest	Greatest
128s	320	160	450
128f	430	140	830
192s	330	210	460
192f	377	220	810
256s	407	310	480
256f	533	300	850

Table 5 reports, for each scheme parametrization, the number of free unrecoverable bits u (the remaining entropy of Table 2), the number of correlation polarity guesses $W = n \cdot 8/w_{\text{arch}}$ (here $w_{\text{arch}} = 32$), the resulting total entropy $u + W$, and the measured recovery time. All `s` variant recoveries used 72 cores of a server with four Intel® Xeon® Gold 6254 CPUs @ 3.10 GHz; the reported time is the wall-clock time until the first matching candidate is found, as the search stops on the first match.

Table 5: Exhaustive search entropy and recovery time per scheme parametrization. Times are time-to-first-match for the six attack procedures; the full search space is 2^{entropy}.

Version	cores	u	W	Entropy $u + W$	Recovery
128f	1	2	4	6	immediately
192f	1	3	6	9	immediately
256f	1	0	8	8	immediately
128s	72	19	4	23	0.3 s
192s	72	29	6	35	223.5 s
256s	72	31	8	39	2704.1 s $\approx$ 45 min

For the three `f` scheme parametrizations, the correct seed is found immediately on a single core. In particular, with `256f`, the CPA already recovers every *bit* of **SK**.seed, so only the $W = 8$ correlation polarities remain, a 2^8 search resolved immediately. For the `128s` variant, the residual entropy is small (≤ 23 bits) and the correct seed is found almost immediately (0.3 s) on 72 cores. The two largest spaces are the `s` scheme parametrizations with $n \in \{24, 32\}$: `192s` has a 2^{35} space and the seed is recovered in 223.5 s on 72 cores. The largest, `256s`, has a 2^{39} space—the relationship of Table 2 brings the otherwise 2^{40} down by one bit—and is recovered in ≈ 45 min on 72 cores. We stress that the reported times are times-to-success and therefore depend on where the true key falls in the enumeration order; the entropy column gives the size of the full search space, which bounds the worst case. Our server was capable of testing approximately 70 million candidates per second, and further optimization can likely be achieved.

6 Conclusion and Countermeasures

In this paper, we have presented the first practical non-profiled SCA on the PRF of SLH-DSA-SHAKE. First, we provided an analysis of the differential SCA vulnerability of SLH-DSA-SHAKE, by carefully examining the inputs of the PRF. Then, we use our analysis to propose a CPA attack with an integrated averaging step, a reduced-complexity CPA, and an efficient remaining exhaustive search algorithm. Finally, we presented concrete results of our attack on simulations and on a real microcontroller. Our work provides a brick for building on the physical security of SLH-DSA, giving implementers information about the feasibility of non-profiled attacks, and the paramount necessity of dedicated side-channel countermeasures.

6.1 Further work

As a further work, we propose the comparison of an attack using the χ step with our attack. As we argued before, attacking the χ step prevents our averaging technique, but it may provide an advantage due to its non-linear property.

Another possible research path is the integration of multiple selection functions. In fact, the same segments of **SK**.seed interact in multiple possible selection function. The leakage of these common selection functions may be integrated to increase the available information for an attacker.

In our work, we argue that for a differential SCA, the usage of the PRF leakage at layer 0 is optimal for input variability. Nevertheless, **SK**.seed still interacts with the upper layer calls of the PRF. Future work could find ways of integrating this leakage in an SCA.

Finally, our attack could be ported into a KECCAK hardware-accelerated SLH-DSA. Parallelism of operations may be a challenge for our attack and may prove to be a valuable contribution for the security of hardware implementations.

6.2 Countermeasures

The most radical countermeasure against our proposed differential side-channel attacks on SLH-DSA is to **restructure the PRF** in the signature generation of SLH-DSA. As argued by Saarinen [Saa24], the only real requirement for the PRF outputs is that they must be deterministic and dependent on the address in the tree. Moreover, if the signature format is not modified elsewhere, a modification of the PRF would be invisible for the verifier, that has only access to the authentication path. Thus, it may be interesting to deviate from the FIPS specification so that the PRF calls do not depend on a single secret seed. For instance, Fluhrer [Flu24] restructures the PRF derivation into a downwards tree of PRF calls rooted at the master **SK**.seed, where nodes feed secret values to WOTS$^+$ and FORS, and generate children nodes, so that the master **SK**.seed is only used directly to derive the first nodes. Each node is used then in a limited number of contexts, reducing exposure to differential side-channel attacks. Of course, this countermeasure would not be standard-compliant, but it completely removes the root cause of our attack.

Masking increases the effort of an attacker, by forcing them to consider higher order attacks. It can provably mitigate differential side-channel attacks on the PRF, under certain attack models. A masking scheme for KECCAK has been analysed by the KECCAK team themselves [BDPA10] and multiple further works [ABP$^+$18,ZSS$^+$21]. Other works have already undertaken the effort of masking the PRF of SLH-DSA. On the hardware side, Saarinen [Saa24] protects the KECCAK primitive with a threshold implementation [NRR06] covering every single PRF call. On software, Banegas and Caullery [BC23] mask the KECCAK permutation throughout SLH-DSA signing on a platform with dedicated vector extensions, but the performance overhead is deemed "too expensive for realistic deployment".

Finally, **shuffling** can reduce the criticality of our attack without preventing it completely. Such a countermeasure can be applied on the internal hash function. For instance, to mitigate their attack on BLAKE-256-based SPHINCS, Kannwischer et al. [KGB$^+$18] suggest shuffling the order of `Mix` calls within a BLAKE-256 round. Such a strategy of shuffling may of course be possible for the KECCAK permutation. Nevertheless, regarding differential side-channel attacks, shuffling by itself only decreases the signal-to-noise ratio at a certain time sample, and it should only serve the purpose of strengthening other countermeasures such as masking.

Acknowledgements

This work has received funding from the EU Horizon Europe research and innovation programme under grant agreement No. 101225722 (FORTRESS). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[ABP$^+$18] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. Rhythmic Keccak: SCA security and low latency in HW. *IACR TCHES*, 2018(1):269–290, 2018.

[BC23] Gustavo Banegas and Florian Caullery. Multi-armed sphincs$^+$. In Jianying Zhou, Lejla Batina, Zengpeng Li, Jingqiang Lin, Eleonora Losiouk, Suryadipta Majumdar, Daisuke Mashima, Weizhi Meng, Stjepan Picek, Mohammad Ashiqur Rahman, Jun Shao, Masaki Shimaoka, Ezekiel O. Soremekun, Chunhua Su, Je Sen Teh, Aleksei Udovenko, Cong Wang, Leo Yu Zhang, and Yury Zhauniarovich, editors, *Applied Cryptography and Network Security Workshops - ACNS 2023 Satellite Workshops, ADSC, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, SiMLA, Kyoto, Japan, June 19-22, 2023, Proceedings*, volume 13907 of *Lecture Notes in Computer Science*, pages 500–514. Springer, 2023.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, Berlin, Heidelberg, August 2004.

[BDH$^+$24] Manuel Barbosa, François Dupressoir, Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. A tight security proof for SPHINCS$^+$, formally verified. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part IV*, volume 15487 of *LNCS*, pages 35–67. Springer, Singapore, December 2024.

[BDPA10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Building power analysis resistant implementations of keccak. In *SHA-3 Conference*, 2010.

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

[BHK$^+$19b] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ signature framework. Cryptology ePrint Archive, Report 2019/1086, 2019.

[BPP$^+$25] Jeremy Boy, Antoon Purnal, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. SLasH-DSA: Breaking SLH-DSA using an extensible end-to-end rowhammer framework. arXiv preprint 2509.13048, 2025.

[CL18] Chun-Yi Chu and Marcin Lukowiak. Two step power attack on SHA-3 based MAC. In *25th International Conference "Mixed Design of Integrated Circuits and System", MIXDES 2018, Gdynia, Poland, June 21-23, 2018*, pages 209–214. IEEE, 2018.

[CLD$^+$26] Yang Cao, Jun Li, Zhibo Du, Kang Yang, and Huixin Li. Correlation power analysis on SLH-DSA algorithm. In *Modern Management based on Big Data VII*, volume 424 of *Frontiers in Artificial Intelligence and Applications*, pages 553–558. IOS Press, 2026.

[CMP18] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. Grafting trees: A fault attack against the SPHINCS framework. In Tanja Lange and Rainer Steinwandt, editors, *PQCrypto 2018*, pages 165–184. Springer, Cham, 2018.

[DRC$^+$25] Patrik Dobias, Azade Rezaeezade, Łukasz Chmielewski, Lukas Malina, and Lejla Batina. SoK: Reassessing side-channel vulnerabilities and countermeasures in PQC implementations. Cryptology ePrint Archive, Report 2025/1222, 2025.

[Flu24] Scott Fluhrer. Side channel resistant sphincs+. Cryptology ePrint Archive, Report 2024/500, 2024.

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

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

[HK22] Andreas Hülsing and Mikhail A. Kudinov. Recovering the tight security proof of SPHINCS$^+$. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part IV*, volume 13794 of *LNCS*, pages 3–33. Springer, Cham, December 2022.

[KGB$^+$18] Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. Differential power analysis of XMSS and SPHINCS. In Junfeng Fan and Benedikt Gierlichs, editors, *COSADE 2018*, volume 10815 of *LNCS*, pages 168–188. Springer, Cham, April 2018.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on Keccak. *IACR TCHES*, 2020(3):243–268, 2020.

[Lam79] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979.

[LCD$^+$26] Xin Li, Yang Cao, Zhibo Du, Fan Zhang, and Jing Peng. Correlation power analysis against the secret key seed of SLH-DSA. In *Modern Management based on Big Data VII*, volume 424 of *Frontiers in Artificial Intelligence and Applications*, pages 648–653. IOS Press, 2026.

[LFF$^+$15] Pei Luo, Yunsi Fei, Xin Fang, A. Adam Ding, David R. Kaeli, and Miriam Leeser. Side-channel analysis of mac-keccak hardware implementations. In Ruby B. Lee, Weidong Shi, and Jakub Szefer, editors, *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2015, Portland, OR, USA, June 14, 2015*, pages 1:1–1:8. ACM, 2015.

[Nat15a] National Institute of Standards and Technology. Secure Hash Standard (SHS), 2015.

[Nat15b] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015.

[Nat24] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard, 2024.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Berlin, Heidelberg, December 2006.

[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, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 276–304. Springer, Cham, August 2024.

[THM+07] Michael Tunstall, Neil Hanley, Robert P McEvoy, Claire Whelan, Colin C Murphy, and William P Marnane. Correlation power analysis of large word sizes. In *IET Irish Signals and Systems Conference (ISSC)*, pages 145–150, 2007.

[TS13a] Mostafa M. I. Taha and Patrick Schaumont. Differential power analysis of mac-keccak at any key-length. In Kazuo Sakiyama and Masayuki Terada, editors, *Advances in Information and Computer Security - 8th International Workshop on Security, IWSEC 2013, Okinawa, Japan, November 18-20, 2013, Proceedings*, volume 8231 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2013.

[TS13b] Mostafa M. I. Taha and Patrick Schaumont. Side-channel analysis of mac-keccak. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*, pages 125–130. IEEE Computer Society, 2013.

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 282–296. Springer, Berlin, Heidelberg, December 2014.

[YK20] Shih-Chun You and Markus G. Kuhn. A template attack to reconstruct the input of SHA-3 on an 8-bit device. In Guido Marco Bertoni and Francesco Regazzoni, editors, *COSADE 2020*, volume 12244 of *LNCS*, pages 25–42. Springer, Cham, April 2020.

[YK21] Shih-Chun You and Markus G. Kuhn. Single-trace fragment template attack on a 32-bit implementation of keccak. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, volume 13173 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2021.

[ZKSH12] Michael Zohner, Michael Kasper, Marc Stöttinger, and Sorin A. Huss. Side channel analysis of the sha-3 finalists. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1012–1017, 2012.

[ZSS+21] Sara Zarei, Aein Rezaei Shahmirzadi, Hadi Soleimany, Raziyeh Salarifard, and Amir Moradi. Low-latency Keccak at any arbitrary order. *IACR TCHES*, 2021(4):388–411, 2021.