Accelerating SLH-DSA by Two Orders of Magnitude with a Single Hash Unit

Markku-Juhani O. Saarinen

SoC Hub Research Centre, Tampere University, Finland
markku-juhani.saarinen@tuni.fi

Abstract. We report on efficient and secure hardware implementation techniques for the FIPS 205 SLH-DSA Hash-Based Signature Standard. We demonstrate that very significant overall performance gains can be obtained from hardware that optimizes the padding formats and iterative hashing processes specific to SLH-DSA. A prototype implementation, SLotH, contains Keccak/SHAKE, SHA2-256, and SHA2-512 cores and supports all 12 parameter sets of SLH-DSA. SLotH also supports side-channel secure PRF computation and Winternitz chains. SLotH drivers run on a small RISC-V control core, as is common in current Root-of-Trust (RoT) systems.

The new features make SLH-DSA on SLotH many times faster compared to similarly-sized general-purpose hash accelerators. Compared to unaccelerated microcontroller implementations, the performance of SLotH's SHAKE variants is up to 300× faster; signature generation with 128f parameter set is is 4,903,978 cycles, while signature verification with 128s parameter set is only 179,603 cycles. The SHA2 parameter sets have approximately half of the speed of SHAKE parameter sets. We observe that the signature verification performance of SLH-DSA's "s" parameter sets is generally better than that of accelerated ECDSA or Dilithium on similarly-sized RoT targets. The area of the full SLotH system is small, from 63 kGE (SHA2, Cat 1 only) to 155 kGe (all parameter sets). Keccak Threshold Implementation adds another 130 kGE.

We provide sensitivity analysis of SLH-DSA in relation to side-channel leakage. We show experimentally that an SLH-DSA implementation with CPU hashing will rapidly leak the SK.seed master key. We perform a 100,000-trace TVLA leakage assessment with a protected SLotH unit.

Keywords: FIPS 205 SLH-DSA · SPHINCS+ · Root-of-Trust · Side-Channel Security

1 Introduction

A Root of Trust (RoT) is a component that forms the basis for the security of an SoC (System on Chip.) An SoC is a large-scale integrated circuit that implements much of the functionality of a typical computer or a similar device. RoT provides cryptographic functions and other features required for secure boot and maintenance of platform security. An important part of this functionality is the verification and generation of digital signatures for integrity checks, authentication, and device attestation.

The U.S. Government has made firmware signatures a priority in their Post-Quantum Cryptography transition timetables [27]. Firmware signature verification is one of the key functions implemented by RoT systems. Hash-based signatures are often viewed as an appropriate, "conservative" selection for such applications, as their security is based simply on the security of hash functions.

Hence many RoT systems already support hash-based signatures in their first-stage boot process. This includes prominent open-source RoTs OpenTitan⋆ and Caliptra⋆⋆. OpenTitan supports SPHINCS$^+$[4] signature verification, while Caliptra supports LMS[18,11] verification. We note that those implementations currently rely on general-purpose hash accelerators, and do not achieve the performance and functionality of the accelerators presented in this work.

The Initial Public Draft (IPD) for FIPS 205, the Stateless Hash-Based Digital Signature Standard (SLH-DSA), was published in August 2023 [26]. In this work, we refer to the standard draft [26] as SLH-DSA. SLH-DSA is derived (with minor modifications) from the SPHINCS$^+$ v3.1 algorithm [4], which was selected as one of the NIST PQC competition winners in July 2022 [1]. SPHINCS$^+$, in turn, is built on a large body of prior research [8].

⋆ OpenTitan silicon root of trust (RoT). https://opentitan.org/

⋆⋆ Caliptra: A Datacenter System on a Chip (SoC) Root of Trust. https://github.com/chipsalliance/Caliptra

1.1 Our Contributions

We provide a quantitative analysis of SLH-DSA and its suitability for RoT applications. We describe SLotH, an open-source implementation that supports all FIPS 205 IPD [26] parameter sets with similar optimization levels, allowing for speed and area comparisons. We observe that SLH-DSA "s" parameter signature verification is faster than that of ECDSA or Dilithium RoT accelerators, indicating good suitability of SLH-DSA for firmware verification.

A general-purpose hash accelerator will speed up an SLH-DSA implementation by a large factor, roughly 10×. Due to message formatting overhead, this will still leave the core hash unit underutilized. As a practical contribution, we show how a "second order of magnitude" (100×) speed-up can be achieved by offloading SLH-DSA-specific Winternitz iteration and key management into hardware and optimizing the firmware. Full hardware and firmware source code is freely available: `https://github.com/slh-dsa/sloth`

While the fragility of SPHICS+ in relation to fault attacks has been previously analyzed [10,3,14,30], masked or side-channel protected implementations of the current SLH-DSA have not been reported [16]. We perform a variable sensitivity analysis and observe that the use of the master secret SK.seed in thousands of PRF calls makes CPU-based implementations highly vulnerable. We verify this experimentally and also perform a 100,000-trace TVLA leakage assessment with a masked (TI) SHAKE256 instantiation that protects the PRF secrets and sensitive hash-chaining operations.

2 Overview and Observations on SLH-DSA

The security of SLH-DSA is based on the second-preimage resistance of hash functions in the SHA-2 (FIPS 180-4) [21] and SHA-3 (FIPS 202) [22] families. These hash functions are used to instantiate six SLH-DSA functions shown in Fig. 1. More precisely, the existential unforgeability (EUF-CMA) security proofs of SLH-DSA in [8,4] require functions with various properties: Pseudorandomness (PQ-PRF), interleaved target subset resilience (PQ-ITSR), and distinct-function multi-target second-preimage resistance (PQ-DM-SPR). Avoidance of a general collision resistance requirement allows the scheme to keep the hash sizes (parameter n) equivalent to the security level in most cases, reducing the signature size and increasing overall efficiency. Under these assumptions, the scheme is designed to provide $Q = 2^{64}$ signatures for each key pair without the need to maintain knowledge of already generated signatures.

2.1 SLH-DSA Parameter Sets

From the SPHINCS$^+$ proposal [4], only the "simple" variants were selected into SLH-DSA, and some relatively minor internal changes were made.

Table 1 contains the parameter sets chosen for the SLH-DSA draft standard, together with its key and signature sizes. The main parameters are:

n : Security parameter/hash length (bytes).
h : Height of the XMSS hypertree.
d : Number of layers in the hypertree. $h' = h/d$ is the layer height.
a : Height of FORS tree; the number of leaves is $t = 2^a$.
k : Number of trees in FORS.
w : Winternitz parameter. For SLH-DSA we have $\lg_2 w = 4$.
m : Message digest length (bytes).

SLH-DSA has $2 \times 3 \times 2 = 12$ parameter sets in total, denoted:

SLH-DSA-{SHA2, SHAKE}-{128,192,256}{s, f}

Here SHA2 and SHAKE are the two hash function families supported. There are three different core hash lengths: $8n \in \{128, 192, 256\}$ bits. These also map to the targeted post-quantum security categories $\{1, 3, 5\}$, respectively. Furthermore, two variants are provided; a small ("s"), and a fast ("f") one. The "s" variants have smaller signature sizes, while the "f" variants require fewer hashes to be computed during signing, making them faster. However, signature verification with "s" variants is faster than with "f" variants; see Table 2.

Hash Function Instantations:

$\mathbf{H_{msg}}(R, \mathsf{PK.seed}, \mathsf{PK.root}, M)$	*(PQ-ITSR)* Instantiated in:
$= \text{SHAKE256}(R \parallel \text{PK} \parallel M, 8m)$	*SHAKE, all*
$= \text{MGF1-SHA-256}(R \parallel \mathsf{PK.seed} \parallel \text{SHA-256}(R \parallel \text{PK} \parallel M), m)$	*SHA2,* $n = 16$
$= \text{MGF1-SHA-512}(R \parallel \mathsf{PK.seed} \parallel \text{SHA-512}(R \parallel \text{PK} \parallel M), m)$	*SHA2,* $n \geq 24$

$\mathbf{PRF}(\mathsf{PK.seed}, \mathsf{SK.seed}, \mathsf{ADRS})$	*(PQ-PRF)* Instantiated in:
$= \text{SHAKE256}(\mathsf{PK.seed} \parallel \mathsf{ADRS} \parallel \mathsf{SK.seed}, 8n)$	*SHAKE, all*
$= \mathsf{Trunc}_n(\text{SHA-256}(\mathsf{PK.seed} \parallel \mathsf{toByte}(0, 64-n) \parallel \mathsf{ADRS}^c \parallel \mathsf{SK.seed}))$	*SHA2, all*

$\mathbf{PRF_{msg}}(\mathsf{SK.prf}, opt\_rand, M)$	*(PQ-PRF)* Instantiated in:
$= \text{SHAKE256}(\mathsf{SK.prf} \parallel opt\_rand \parallel M, 8n)$	*SHAKE, all*
$= \mathsf{Trunc}_n(\text{HMAC-SHA-256}(\mathsf{SK.prf}, opt\_rand \parallel M))$	*SHA2,* $n = 16$
$= \mathsf{Trunc}_n(\text{HMAC-SHA-512}(\mathsf{SK.prf}, opt\_rand \parallel M))$	*SHA2,* $n \geq 24$

$\mathbf{F}(\mathsf{PK.seed}, \mathsf{ADRS}, M_1)$	*(PQ-DM-SPR)* Instantiated in:
$= \text{SHAKE256}(\mathsf{PK.seed} \parallel \mathsf{ADRS} \parallel M_1, 8n)$	*SHAKE, all*
$= \mathsf{Trunc}_n(\text{SHA-256}(\mathsf{PK.seed} \parallel \mathsf{toByte}(0, 64-n) \parallel \mathsf{ADRS}^c \parallel M_1))$	*SHA2, all*

$\mathbf{H}(\mathsf{PK.seed}, \mathsf{ADRS}, M_2)$	*(PQ-DM-SPR)* Instantiated in:
$= \text{SHAKE256}(\mathsf{PK.seed} \parallel \mathsf{ADRS} \parallel M_2, 8n)$	*SHAKE, all*
$= \mathsf{Trunc}_n(\text{SHA-256}(\mathsf{PK.seed} \parallel \mathsf{toByte}(0, 64-n) \parallel \mathsf{ADRS}^c \parallel M_2))$	*SHA2,* $n = 16$
$= \mathsf{Trunc}_n(\text{SHA-512}(\mathsf{PK.seed} \parallel \mathsf{toByte}(0, 128-n) \parallel \mathsf{ADRS}^c \parallel M_2))$	*SHA2,* $n \geq 24$

$\mathbf{T}_\ell(\mathsf{PK.seed}, \mathsf{ADRS}, M_\ell)$	*(PQ-DM-SPR)* Instantiated in:
$= \text{SHAKE256}(\mathsf{PK.seed} \parallel \mathsf{ADRS} \parallel M_\ell, 8n)$	*SHAKE, all*
$= \mathsf{Trunc}_n(\text{SHA-256}(\mathsf{PK.seed} \parallel \mathsf{toByte}(0, 64-n) \parallel \mathsf{ADRS}^c \parallel M_\ell))$	*SHA2,* $n = 16$
$= \mathsf{Trunc}_n(\text{SHA-512}(\mathsf{PK.seed} \parallel \mathsf{toByte}(0, 128-n) \parallel \mathsf{ADRS}^c \parallel M_\ell))$	*SHA2,* $n \geq 24$

Lengths of variables in bytes:

User message $|M|$ = any length. $\text{PK} = (\mathsf{PK.seed} \parallel \mathsf{PK.root}), |\text{PK}| = 2n$.
$|\mathsf{SK.seed}| = |\mathsf{SK.prf}| = |\mathsf{PK.seed}| = |\mathsf{PK.root}| = |R| = |M_1| = n$, $|M_2| = 2n$,
$|M_\ell| \in \{\mathsf{len} * n, kn\}$. $|opt\_rand| = n$, $|\mathsf{ADRS}| = 32$, $|\text{ADSR}^c| = 22$.

Subfunctions:

$\mathsf{toByte}(0, n)$:	A sequence of n zero bytes.
$\mathsf{Trunc}_n(X)$:	First n bytes of string X (truncation).
$\text{SHAKE256}(M, 8n)$:	First n bytes from SHAKE256 XOF (FIPS 202) [22].
$\text{SHA-256}(X)$:	32-byte hash result from SHA2-256 (FIPS 180-4) [21].
$\text{SHA-512}(X)$:	64-byte hash result from SHA2-512 (FIPS 180-4) [21].
$\text{MGF1-SHA-256}(X, m)$:	First m bytes from MGF "counter mode", SHA2-256 [5].
$\text{MGF1-SHA-512}(X, m)$:	First m bytes from MGF "counter mode", SHA2-512 [5].
$\text{HMAC-SHA-256}(K, X)$:	32-byte HMAC of X with key K using SHA2-256 [20].
$\text{HMAC-SHA-512}(K, X)$:	64-byte HMAC of X with key K using SHA2-512 [20].

Fig. 1. Hash function instantations in SLH-DSA. The main formats are directly supported by SLotH hardware. Keys PK.seed, SK.seed, and address variable ADRS are held in special registers for automatic formatting and padding.

Table 1. SLH-DSA parameter sets. SHA2 and SHAKE variants are identical apart from hash function instantiations, and have the same signature lengths |SIG|. The secret (signing) key SK includes the public (verification) key PK.

Parameter set.	*PQ*	*Internal parameters.*								*Sizes in bytes.*		
SHA2 or SHAKE	Sec	n	h	d	h'	a	k	w	m	|PK|	|SK|	|SIG|
SLH-DSA-*-128s	1	16	63	7	9	12	14	16	30	32	64	7,856
SLH-DSA-*-128f	1	16	66	22	3	6	33	16	34	32	64	17,088
SLH-DSA-*-192s	3	24	63	7	9	14	17	16	39	48	96	16,224
SLH-DSA-*-192f	3	24	66	22	3	8	33	16	42	48	96	35,664
SLH-DSA-*-256s	5	32	64	8	8	14	22	16	47	64	128	29,792
SLH-DSA-*-256f	5	32	68	17	4	9	35	16	49	64	128	49,856

SLH-DSA Keys. In SLH-DSA, the public (signature verification) key PK has two n-byte components, while the signing key SK has two additional secret n-byte components.

$$\text{PK} = (\mathsf{PK.seed}, \mathsf{PK.root}) \tag{1}$$

$$\text{SK} = (\mathsf{SK.seed}, \mathsf{SK.prf}, \mathsf{PK.seed}, \mathsf{PK.root}) \tag{2}$$

All components except PK.root are random (generated with an RBG). We won't go into details of key generation, but it is a relatively straightforward process: PK.root is the root of the final layer of the XMSS hypertree (Section 2.4) and always recomputed in signature verification for comparison.

Signature format. An SLH-DSA signature has three main parts, each created in a distinct step of the signing process:

$$\text{SIG} = R \parallel \text{SIG}_{FORS} \parallel \text{SIG}_{HT} \tag{3}$$

The R randomizer (Section 2.2) is n bytes, the SIG_{FORS} (Section 2.3) component is kan bytes, while SIG_{HT} (Section 2.4) is $(h + d * \mathsf{len})n$ bytes. We give a high-level view of the signing process (function slh_sign [26, Alg. 18]) here, together with analysis and commentary related to implementation and security aspects.

2.2 *R*: Randomized Hashing

The signing process starts with a randomized hashing of the message to be signed. A two-pass mechanism is used; the first pass derives the randomizer R (Eq. 4), and the second one (Eq. 5) produces a digest that is actually signed.

$$R \leftarrow \mathbf{PRF_{msg}}(\mathsf{SK.prf}, \mathit{opt\_rand}, M) \tag{4}$$

$$\mathit{digest} \leftarrow \mathbf{H_{msg}}(R, \mathsf{PK.seed}, \mathsf{PK.root}, M) \tag{5}$$

$$\mathit{md} \parallel \mathit{idx} \leftarrow \mathit{digest} \tag{6}$$

The verification process only needs a single pass with $\mathbf{H_{msg}}$ (Eq. 5). It is easy to see that Eq. (4) is essentially redundant in signing if a secure RBG is available to reliably produce a random R. SLotH follows the two-pass flow of the SLH-DSA specification in this aspect, mainly for compliance reasons.

The m-byte md component split from the digest (Eq. 6) is signed in the FORS step (Section 2.3). The h-bit component idx specifies the XMSS authentication path and also the FORS public key for the XMSS signature (Section 2.4).

Implementation Analysis. The randomized hashing step is the only one accessing the user-supplied message M itself. It is possible that the $\mathbf{H_{msg}}$ (and $\mathbf{PRF_{msg}}$) are computed outside the SLH-DSA module (e.g. by the system main CPU) and the *digest* passed to it for signature creation or verification. This way a RoT unit (such as one containing SLotH) may have relatively low-bandwidth interfaces.

For latency-critical verification of large amounts of data (e.g. in the boot process), it is possible to store the *digest* in the data header and start signature verification before the hash has been actually computed. The verifier checks afterward that $\mathbf{H_{msg}}$ has indeed produced the correct value.

From a side-channel perspective, the R randomizer generation (Eq. 4) is the only part handling confidential variables (SK.prf) in this step. However, SK.prf is essentially redundant if randomization is used. Implementations may choose to not even store SK.prf.

The impact of faulting the randomized hashing step is that a signature for some other message may be produced. Given that in the security model, the adversary can query up to Q signatures at will, the risks from faulting this step appear to be lower than from the subsequent hashes.

2.3 SIG_{FORS}: FORS Signature of the Message

SLH-DSA uses the few-time signature scheme FORS to sign the message M itself. FORS is "few-time" in relation to a specific FORS secret key. For each SLH-DSA keypair, there are 2^h possible FORS keys, and one is chosen pseudorandomly (using h-bit idx) for each message. Since $h \geq 63$ for SLH-DSA parameters, a statistical argument shows that the security risk of FORS key re-use remains within bounds for up to $Q = 2^{64}$ signatures. After this, there is a gradual increase in the risk of signature forgery.

Fig. 2 illustrates the FORS signing process. The secret keys of the FORS scheme are the 2^a leaf nodes for each of the k FORS trees; these are dynamically generated with **PRF** using the master secret SK.seed and tree index idx (fors_SKgen). In signing, the message digest component md is split into a-bit chunks. Each of those is used to select a leaf node (index) in one of the k Merkle trees. The authentication paths from leaf nodes to respective roots form the SIG_{FORS} signature. The concatenation of k root nodes is hashed, and this is the SIG_{PK} public key. In verification, the signature is valid if hashing the preimage paths provided in SIG_{FORS} leads the same roots and hence to PK_{FORS}.

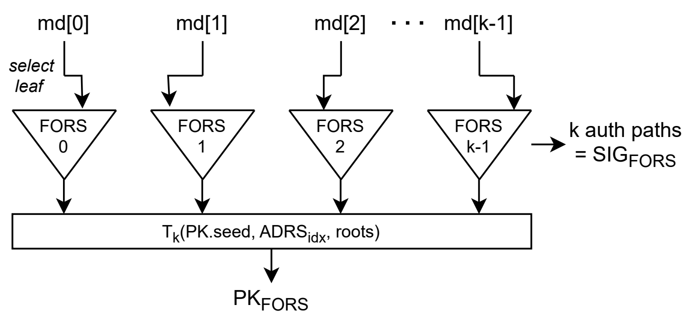

Fig. 2. Each SLH-DSA keypair defines 2^h pseudorandom FORS keypairs; idx selects which one is used. FORS few-time signature generation uses a-bit segments of the message digest md to select leaf nodes in k Merkle trees. The trees have height a and are illustrated here with roots at the bottom. The authentication paths (hashes) to reach the roots from the leaves form the message signature SIG_{FORS}.

Implementation Analysis. The FORS step can be k-way parallelized in relation to the trees, as those computations are independent, apart from the final $\mathbf{T}_\ell$ that combines the k tree roots into final PK_{FORS}. In a sequential implementation one can randomize the execution order in relation to the trees as an inexpensive side-channel countermeasure, and to de-synchronize against a targeted FIA. It is also possible to randomize the tree traversal order (Alg. 14, fors_node in [26]).

The FORS signing step uses **PRF** to generate the leaf secret keys, and an additional **F** iteration to bind the secret key with a message-dependent index. For side-channel protection, one needs to put extra effort into protecting **PRF** as it directly uses the master secret SK.seed. Our protected implementation masks **PRF** and also the following **F** binding step (Alg. 13, fors_SKgen everywhere and also **F** on line 8 of Alg. 14, fors_node in [26].)

Errors from faulting almost any hash of the FORS signing step will cause a hash path and a PK_{FORS} public key which is completely different from the correct one. However, since the subsequent XMSS step (Section 2.4) simply authenticates this value, the produced SLH-DSA signature will still be verified as a valid one. Hence a signature verification check does not protect against fault attacks in the case of the SLH-DSA signing process. Furthermore, as has been observed (and exploited) by Genêt [14] and others, the faulty SIG_{FORS} reveals information that can be used to forge signatures with high likelihood.

2.4 SIG_{HT}: XMSS Hypertree Signature of the FORS Public Key

The final step in the SLH-DSA signing process authenticates the PK_{FORS} public key using an XMSS hypertree. The hypertree supports a total of 2^h one-time Winternitz signatures, which are structured into d layers. Each of the XMSS tree layers has height $h' = h/d$ and contains WOTS public keys in its $2^{h'}$ leaf nodes. WOTS signatures authenticate the root (public key) of the previous layer, or PK_{FORS} on the first layer. The final XMSS root is the main SLH-DSA public verification key PK.root. Fig. 3 illustrates the hypertree signing process with a toy example, but we refer to [26] for details and terminology.

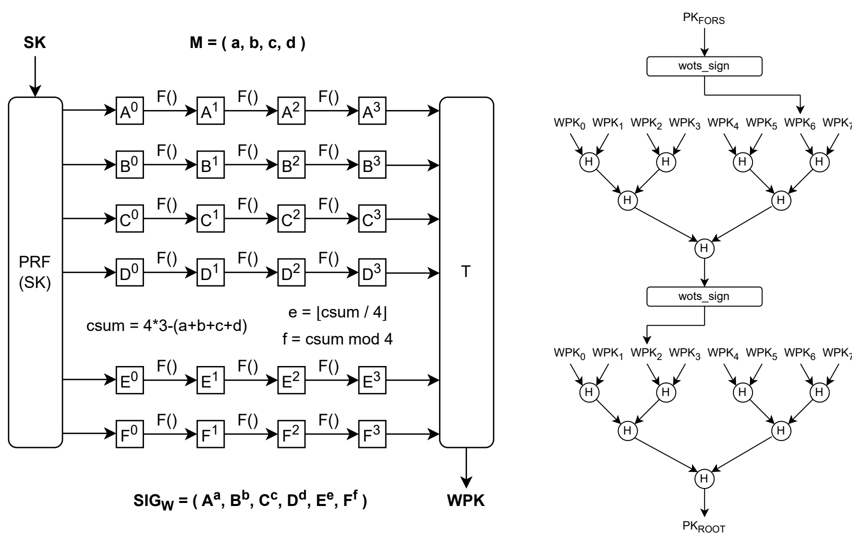

Fig. 3. A toy example of an XMSS hypertree signature, similar to the one in SLH-DSA. We illustrate WOTS (top) by signing an 8-bit message organized into four-bit pairs $M = (a, b, c, d)$. Two additional checksum digits (e, f) are required. The hypertree (bottom) consists of d (here 2) layers of XMSS trees, each with a WOTS one-time signature authenticating the root of the previous layer.

Implementation Analysis. In SLH-DSA key generation and signing, a **PRF** call to generate the WOTS public key is followed by Winternitz chain iteration with function **F**. Simplifying ADRS (domain-separation addressing) syntax, we write:

$$X^0 = \mathbf{PRF}(\mathsf{PK.seed}, \mathsf{SK.seed}, \texttt{WOTS\_PRF}) \tag{7}$$

$$X^j = \mathbf{F}(\mathsf{PK.seed}, \texttt{WOTS\_HASH}(j), X^{j-1}) \quad \text{for } j \geq 1. \tag{8}$$

In key generation (wots_PKgen) the index $X^{w-1} = X^{15}$ is the result, while the signing process (wots_sign) uses $X^{\text{msg}[i]}$. The wots_PKFromSig verification function evaluates Eq. (8) only, between $X^{\text{msg}[i]}$ and X^{15}.

As highlighted in Table 2, the **F** invocations in Winternitz chains (Eq. 8) completely dominate the SLH-DSA computational cost, making up roughly 80% of all of the hash function invocations in signing and 90% in verification. The **PRF** calls of Eq. (7) make up much of the rest of the hash invocations.

Clearly, an implementation should optimize chaining; our implementation moves the padding and iteration of Eq. (8) completely into hardware, in which case there are only 1 or 2 cycles between autonomous hash iterations. A helpful aspect in the case of SHA2 is that all security levels use SHA2-256 for these core operations; the SHA2-512 core does not need this feature.

From a side-channel leakage perspective, the hashes have a decreasing sensitivity. Since the thousands of invocations of **PRF** directly use the master secret key SK.seed, **PRF** requires protection to prevent horizontal attacks against this variable. The sensitivity of X^1 and above is lessened by randomization by idx and other contents of ADRS, but may also require protection. Our protected implementation masks **PRF** everywhere and also subsequent chaining operation in key generation and signing (lines 6–7 of Alg 5., wots_PKgen and lines 17–19 of Alg. 6, wots_sign in [26].)

In the signing process, each layer of the XMSS hypertree computation will authenticate the previous layer (or PK_{FORS} in the case of the initial layer) regardless of whether it is correct or not. As noted in Section 2.3, faulted signatures will be verified as correct, but will reveal sensitive information that can be used to create forgeries and mount other attacks [10,30,14]. Intuitively, the PK_{FORS} index is always the same for a given idx regardless of md; hence the one-timeness of XMSS is not violated in the first layer as the same index is used to sign the same message. However, faulting anywhere in the process will cause XMSS to potentially use the index twice, enabling forgeries. As observed in [14] and other works, redundancy via error detection/correction and repeated computations appears to be the only robust countermeasures, assisted by control flow randomization against targeted faults.

Table 2. Quantitative analysis: Distribution of high-level hash function invocations with standard SLH-DSA parameter sets. The distribution of high-level calls is independent of instantiation (same number for both SHAKE and SHA2, with or without acceleration.) Averages for 2000 runs are given for **F** invocations in signing and verification functions – other functions use a constant number of invocations. The single $\mathbf{H_{msg}}$ and $\mathbf{PRF_{msg}}$ calls are omitted in the table for space but are included in the total. Also listed are the number of chain() calls, the number of **F** calls in chains, and the percentage of these chaining **F**'s of the total number of high-level hash invocations.

Key Generation (slh_keygen)

Funct.	**128f**	**192f**	**256f**	**128s**	**192s**	**256s**
PRF	280	408	1,072	17,920	26,112	17,152
F	4,200	6,120	16,080	268,800	391,680	257,280
H	7	7	15	511	511	255
$\mathbf{T}_\ell$	8	8	16	512	512	256
Total	4,495	6,543	17,183	287,743	418,815	274,943
chain()	280	408	1,072	17,920	26,112	17,152
chain F	4,200	6,120	16,080	268,800	391,680	257,280
chain %	93.4%	93.5%	93.6%	93.4%	93.5%	93.6%

Signature Generation (slh_sign)

Funct.	**128f**	**192f**	**256f**	**128s**	**192s**	**256s**
PRF	8,272	17,424	36,144	182,784	461,312	497,664
F	94,246	142,697	290,775	1,938,676	3,019,898	2,418,182
H	2,230	8,566	18,136	60,898	282,079	362,458
$\mathbf{T}_\ell$	176	176	272	3,584	3,584	2,048
Total	104,926	168,865	345,329	2,185,944	3,766,875	3,280,354
chain()	6,895	10,047	19,296	125,650	183,090	137,685
chain F	92,134	134,249	272,855	1,881,332	2,741,370	2,057,734
chain %	87.8%	79.5%	79.0%	86.1%	72.8%	62.7%

Signature Verification (slh_verify)

Funct.	**128f**	**192f**	**256f**	**128s**	**192s**	**256s**
PRF	0	0	0	0	0	0
F	5,908	8,620	8,633	1,886	2,751	4,067
H	264	330	383	231	301	372
$\mathbf{T}_\ell$	23	23	18	8	8	9
Total	6,196	8,974	9,035	2,126	3,061	4,449
chain()	770	1,122	1,139	245	357	536
chain F	5,875	8,587	8,598	1,872	2,734	4,045
chain %	94.8%	95.7%	95.2%	88.1%	89.3%	90.9%

3 Hardware Architecture

The hardware components of SLotH were newly written for the project, apart from the small "pug" RISC-V core that the author uses for prototyping. Custom instruction set extensions or other special features are not used; the RISC-V core can be replaced by almost any other RV32IMC core, e.g. IBEX*** or VeeR†. There is no reason not to expect that an ARM Cortex M3/M4 core or some other controller type commonly used in RoTs would not work as the amount of assembler code is very small.

Fig. 4 illustrates the relationships between logical components; the CPU is expected to be a generic RoT controller and the SLotH accelerators can be instantiated independently of each other to support various configurations (when there is no need for both SLH-DSA-SHA2 and SLH-DSA-SHAKE, or for masking in a verify-only module.) Since the unit is relatively small (Table 3), redundancy can be provided by duplicating it entirely.

The SHA2-256 unit is sufficient to support SLH-DSA-SHA2-128 variants, while the SHA2-512 unit is additionally required for Category 3 and 5. The Keccak module supports all security levels of SLH-DSA. SCA security is currently only provided for Keccak via a fast Threshold Implementation (TI). This module is very large and not required for signature verification, and hence currently separate from unmasked Keccak.

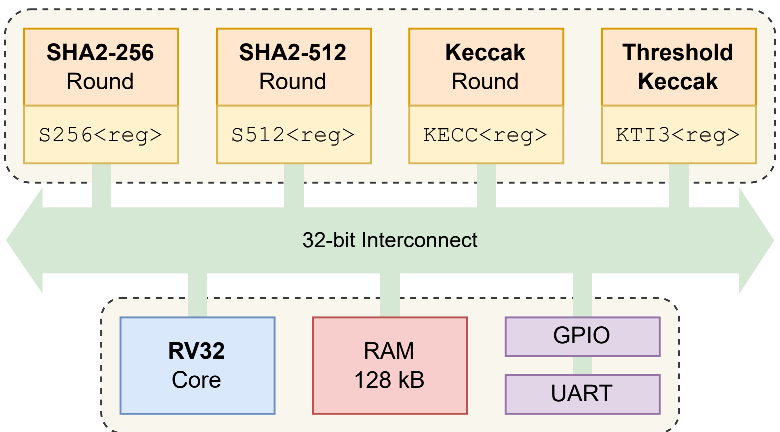

Fig. 4. SLotH consists of independent memory-mapped accelerators (top) and firmware that runs on a generic embedded RISC-V controller (bottom). SHA2-256 and SHA2-512 units are independent and the threshold (side-channel protected) Keccak is a separate unit from the "normal" Keccak. The FPGA instantiations have simple UART and GPIO external interfaces, and 128kB of Block RAM to hold the firmware and work memory (stack). RoT "production" silicon would have different types of interfaces and parts of the firmware would be in mask ROM.

3.1 SLotH Keccak (SHAKE) Units

The Keccak unit computes the Keccak-p[1600, 24] permutation [22] (and single-block SHAKE256 functions **PRF**, **F**, **H**) in 24 cycles at all security levels. The chaining modes require at most 2 additional cycles per Winternitz iteration.

The Keccak accelerator has two components, the Keccak f1600 round function (`keccak_round.v`) and its memory-mapped control logic (`keccak_sloth.v`). The round function is pure combinatorial logic with 1600-bit inputs and outputs for the state. It updates the Keccak round constant LFSR, which also serves as the round counter. The combinatorial logic allows one to potentially combine two round functions into one clock cycle if needed (this is plausible as the Keccak critical path is relatively short).

*** IBEX (`https://github.com/lowRISC/ibex`) is the RV32 core used by the OpenTitan RoT project.

† VeeR (`https://github.com/chipsalliance/Cores-VeeR-EL2`) is the RV32 core used by the Caliptra RoT project.

The control unit maintains a memory-mapped register interface and offers several modes of operation, including automatic padding and iteration for Winternitz chains. See Table 7 for an overview of the control registers. The module also holds the contents of PK.seed, SK.seed, and ADRS variables as they are required to create the message formats for **PRF**, **F**, **H**, $\mathbf{T}_\ell$ functions (Fig. 1.) Due to similarities between SLH-DSA SHAKE message formats, we don't have to implement all of these separately; furthermore quantitative analysis (Table 2) shows that optimization of some formats is more important than others.

Each security level $n \in \{16, 24, 32\}$ (set in the `KECC_SECN` register) changes the sizes of the message fields in the message formats (requiring wide MUXes for fast implementation.) Furthermore, the "hash address" component in ADRS is incremented after each hash iteration during autonomous Winternitz chain operation triggered by a write to the `KECC_CHNS` iteration count register.

The unit supports raw memory-mapped permutation as well, allowing other SHA-3-derived functions and modes of operation to be accelerated. Hence the Keccak module may also be used to provide support for ML-KEM (FIPS 203 Kyber [25]) and ML-DSA (FIPS 204 Dilithium [26]), as about half of the cycles of those algorithms are typically spent on Keccak computation. However, these algorithms benefit from different types of Keccak "helper" optimization features than SLH-DSA.

Threshold Implementation. For side-channel security experiments, a fast (24-cycle) Threshold Implementation [19] is used. As can be seen from the register map Table 7, it is functionally equivalent to the standard Keccak unit, apart from operating with three Boolean shares for the cached secret key and the entire state. Performance reduction in relation to standard Keccak comes mainly from CPU-operated setting up and "collapsing" of masked results.

The threshold implementation technique is very similar to the one originally proposed in [9]. Hence it is lacking in certain theoretical aspects as it does not achieve uniformity; "changing of the guards" [12] or active re-randomization techniques are not used. Furthermore, only the secret key components of the input hashes are refreshed. However, the extremely high speed and parallelism of the implementation help to minimize leakage in practice; see Section 6.2.

3.2 SLotH SHA2-256 and SHA2-512 Units

The SHA2 [21] units process a message block (a full compression function, including simultaneous message schedule) of SHA2-256 and SHA2-512 in 64 cycles and 80 cycles, respectively. The padding and Winternitz iteration features require at most 2 additional cycles.

SHA2 is effectively two different algorithms from a hardware perspective: SHA2-256 has 32-bit state variables while SHA2-512 has 64-bit state variables and a twice larger block size; its round function is more than twice as large (Table 3). While there are obvious similarities between SHA2-256 and SHA2-512, there are performance disadvantages in combining the two implementations. The carry chains in adders make the critical paths of SHA2 rather long – adding MUXes to simultaneously deal with 32-bit and 64-bit nonlinear functions and rotations on the data path would create a bottleneck against high clock frequencies. Area savings from combining the two modules are relatively small.

The SHA2 accelerators are structured into two components: Logic for the compression function and message schedule (`sha256_round.v`, `sha512_round.v`) and memory-mapped controls (`sha256_sloth.v`, `sha512_sloth.v`). The SHA2-256 control unit `sha256_sloth.v` supports automatic padding and Winternitz iteration, while the SHA2-512 version `sha512_sloth.v` does not (See register map in Table 8). This is because all security levels and parameters of SLH-DSA-SHA2 instantiate **F** and **PRF** with SHA2-256. SHA2-512 is not used in chaining or secret key computation (See Fig. 1)

A noteworthy optimization built into the SHA2 instantiations of SLH-DSA itself is the padding of PK.seed into the initial 64 bytes of the input in **PRF**, **F**, **H**, $\mathbf{T}_\ell$ functions. Since 64 bytes is the block size of SHA2-256, there is no need to compute this initial block in each high-level function invocation; one can just store the contents of the 256-bit chaining variable after that block and use it for operations. This "key-dependent IV" is set in the S256_SEED register and automatically used in hash preparation.

The SHA2 control unit is able to handle compressed 22-byte ADRS^c headers automatically, extracting the necessary bytes from the internal holding state S256_ADRS. While the 22-byte ADRS^c allows SLH-DSA to fit a 32-byte M_1 (for **F**) or SK.seed (for **PRF**) into the final SHA2-256 block together with padding, it also creates a performance issue as elements following the 22-byte field are not aligned to word boundaries.

This affects $\mathbf{H}$ and $\mathbf{T}_\ell$ as well. The implementation optimizes this bottleneck by making the message input registers of the SHA2-256 SLotH module available for unaligned writes via an unaligned write mirror.

The secret key SK.seed is provided via the S256_SKSD register for **PRF** computation, where it can be semi-permanently stored. As with the Keccak unit, the control firmware can "forget" the secret key.

The current version of the SHA2-512 control unit `sha512_sloth.v` (Table 8) is substantially simpler than the SHA2-256 unit, as it doesn't need to support Winternitz chaining. Non-aligned offset writes are supported to speed up the formatting of $\mathbf{H}$ and $\mathbf{T}_\ell$ with $n = 24$ and $n = 32$ (security categories 3 and 5). Similarly to the SHA2-256 case, an unaligned write mirror is provided.

3.3 Hardware Complexity and Size

Table 3 summarizes the relative synthesis sizes of submodules on FPGA and ASIC targets. If only Category 1 (SLH-DSA-SHA2-128{s,f}) security parameters are required, then the SHA2-256 accelerator is sufficient. It is very compact. However, we find that to support all security categories, the Keccak (SHAKE) accelerator is smaller than the SHA2 accelerator as Category 3 and 5 instantiations (SLH-DSA-SHA2-{192,256}){s,f} require both SHA2-256 and SHA2-512 modules (See Fig. 1).

To assess audit complexity we note that the implementation discussed in this report is about 3000 lines of Verilog, with the RV32IMC control core and basic IO peripherals adding another 1000 lines. No external IP modules are used.

Mid-range FPGA. The design was targeted on the ChipWhisperer CW305 board[‡] with Xilinx (AMD) Artix-7 chip XC7A100T-2FTG256 (a chip with a list price around $100.) The Artix-7 family has been the de facto evaluation target on mid-range FPGAs in recent NIST LWC and PQC efforts. An all-algorithm test system requires 14,428 LUTs (logic resource utilization 22.76%), while the TI Keccak doubles this to 30,717 LUTs (48.45%). Table 3 lists the relative sizes of the submodules. All synthesized FPGA configurations had 32 Block RAM tiles for 128kB of main memory and were functional systems with interconnect, simple IO components, etc. No DSP units or other special logic was used. Vivado 2023.2 with a 100 MHz timing constraint was used, which is a typical system clock for complex designs instantiated on Artix-7.

High-end FPGA. We also instantiated the design on a higher-end FPGA using the Xilinx VCU118 Evaluation Kit, which has an XCVU9P-L2FLGA2104 chip, belonging to the Virtex UltraScale+ family. Full-system synthesis was targeted at 250 MHz and required 14,237 CLB LUTs, the same amount of block RAM tiles as the Artix-7 version, and some miscellaneous other resources. Utilization of any logic fabric resource didn't exceed 1.61%, so dozens of independent, fully-featured SLotH units can probably be made to run on a single chip (however, we have not performed this experiment.) The Threshold Keccak unit doubles the area to 30,692 CLB LUTs, with maximum utilization of 3.42%.

ASIC Area Estimates. We used the Nangate45 cell library and the Yosys / OpenSTA flow from lowRISC IBEX[§] to estimate ASIC sizes. This flow reports 27.3kGE for the default ("small") configuration of the IBEX core. Table 3 contains synthesis reports for various SLotH configurations. Synthesis settings were kept at IBEX repository defaults, including a 4ns / 250 MHz timing closure, which was met with a significant slack. Tool versions were built from source code in early 2024. The 128kB main memory was excluded from synthesis, but (the rather large) internal holding registers of the accelerators were included.

4 SLotH Firmware

Development process. The core SLH-DSA algorithm implementation of SLotH was created using the FIPS 205 IPD [26] specification – not adapted from prior implementations. We first created a Python model and verified that it matches the FIPS-updated Known Answer Tests (KATs) available from the SPHINCS$^+$ team. We then implemented a portable, stand-alone C version suitable for bare metal targets as well as generic

‡ NewAE/lowRISC CW305 Artix Target: `https://rtfm.newae.com/Targets/CW305%20Artix%20FPGA/`

§ IBEX Yosys/OpenSTA Flow: `https://github.com/lowRISC/ibex/blob/master/syn/README.md`

Table 3. Artix-7 FPGA LUT utilization and estimated Nangate45 silicon area in thousands of NAND2 Gate Equivalents (kGE). Synthesis results for various system configurations; the plus sign (+) indicates the total area increase caused by the accelerator configuration in relation to the control unit on the top row.

CPU+IX RV32IMC	**Keccak** "plain"	**SHA2 -256**	**SHA2 -512**	**Keccak TI3**	**LUTs** XC7A100T	**kGE** Nangate45
yes	-	-	-	-	(3,023)	(31.36)
yes	-	yes	-	-	+2,463	+32.03
yes	yes	-	-	-	+5,582	+41.72
yes	-	yes	yes	-	+5,942	+82.36
yes	yes	yes	-	-	+8.205	+73.52
yes	yes	yes	yes	-	+10,857	+123.99
				Full system, all SLH-DSA parameters:	**14,428**	**155.35**
yes	yes	-	-	yes	+21,826	+173.22
yes	yes	yes	yes	yes	+27,694	+254.48
				Full system with Three-Share TI Keccak:	**30,717**	**285.84**

Table 4. RISC-V Firmware size of SLotH components. One can remove either SHA2 or SHAKE driver components if those are not required. Compiled with gcc version 13.2.0, flags `-O2 -mabi=ilp32 -march=rv32imc`.

bytes	**file (source)**	**Description.**
4,928	`slh/slh_dsa.o`	SLH-DSA algorithm, common for all parameter sets.
6,124	`drv/sloth_sha2.o`	SHA2 parameters and SLotH SHA2-256/512 driver.
3,299	`drv/sloth_shake.o`	SHAKE parameters and SLotH Keccak driver.
681	`slh/sha2_256.o`	Plain SHA2-256 padding / C API.
771	`slh/sha2_512.o`	Plain SHA2-512 padding / C API.
566	`slh/sha3_api.o`	Plain SHA3/SHAKE padding / C API.
16,369	**total**	Complete SLH-DSA binary size, all 12 parameter sets.

PCs. This is still a part of the distribution – one can run SLH-DSA without any hardware acceleration too (software implementations of SHA2 and SHAKE are included.)

The co-design phase to develop hardware drivers was guided by quantitative analysis of SLH-DSA, end-to-end Verilator benchmark tests, and profiling. The prototype system uses the reference KATs as self-tests; we adopted the NIST KAT generator (including its deterministic AES-based randombytes() component) on the target and uses checksums to match keys and signatures. To estimate audit effort, we note that the entire firmware (including self-tests, headers, etc) is about 7,060 lines of C code.

Firmware structure and size. Many PQC algorithm implementations hard-code parameters into C macros, necessitating duplication of the entire implementation for different parameter sets. We wanted the same, compact firmware to be able to run all parameter sets. Hence algorithm parameters are read from an object-like `struct` that also provides a function pointer abstraction ("methods") for the optimized core hash functions. There is a performance penalty to this as function pointers hinder inlining and link-time optimization, but the resulting firmware is much more compact as a result, simultaneously supporting all 12 parameter sets (both SHA2 and SHAKE) in 16.4 kB (Table 4). Note that the optional side-channel countermeasures are in hardware and do not require much additional firmware support.

For each parameter set, the drivers provide the numerical parameters themselves (n, h, d, h', a, k, m), functions for creating a hardware context (including a direct-hardware pointer to an ADRS structure), core hash instantiations ($\mathbf{H_{msg}}$, $\mathbf{PRF}$, $\mathbf{PRF_{msg}}$, $\mathbf{F}$, $\mathbf{H}$, $\mathbf{T}_\ell$). Some more complex primitives are also supported by the drivers: the Winternitz iteration of $\mathbf{F}$ in chain() [26, Alg. 4] in WOTS verification and a combined $\mathbf{PRF} + \mathbf{F}$ operation for FORS key generation and signing [26, Algs. 5 and 6]. All instances of $\mathbf{PRF}$ utilize hardware-stored secret keys and automatic formatting (See Section 6.4.)

The HAL used in the SLotH prototype is very primitive. As a closely coupled bare metal embedded system with a dedicated MCU, the "drivers" directly poke the control registers and wait for completion before issuing new commands. This direct control helps to reduce latency and hence to maximize the utilization of the hash units. Note that the entire SHAKE block processing time is 24 cycles, which is less than the typical interrupt handler latency alone. Memory copying is minimized in critical sections by hardware formatting features, such as the direct ADRS registers.

RAM Usage. The SLH-DSA signing process does not require much working memory beyond that for the signature itself, which can be almost 50 kB in size (Table 1). As noted in Section 2.2, there are potential techniques to externalize the computation of message hashing outside the SLH-DSA module itself. Unlike software countermeasures, hardware side-channel countermeasures do not significantly increase the RAM footprint. We measured the maximum additional stack depth (temporary working memory usage) of the SLH-DSA primitives to be 3,956 bytes required by the SLH-DSA-SHA2-256s signing function.

5 Performance Analysis

Table 5 contains end-to-end measurements of SLotH cycle counts for the high-level functions in FIPS 205 IPD [26]: Key Generation (slh_keygen, Alg. 17), Signature Generation (slh_sign, Alg. 18), Signature Verification (slh_verify, Alg. 19.) A short/single-block message M used for Sign and Verify benchmarking. All 12 variants tested pass a Known Answer Test comparison with an updated ("post-FIPS") reference implementation of SPHINCS$^+$. The benchmarked implementations didn't deploy SCA or FIA countermeasures (See Table 6.)

As the algorithms do not interact with platform-dependent components such as external memories, the cycle counts are the same for all synthesis targets (XC7A100T @ 100 MHz, XCVU9P @ 250 MHz, Nangate45 @ 250 MHz). There is no reason to expect that higher-end technology nodes will not support substantially higher clock frequencies for this design.

Table 5. Clock cycles for the current version of SLotH in end-to-end testing (average of 100 iterations.) We also include a clk/h metric, where we divide cycles by the number of high-level hash invocations (Table 2). We compute the same metric for Cortex M4 benchmarks from PQM4 [17] to illustrate the effect of custom SLH-DSA acceleration. For Cortex M4, its 12,000-cycle Keccak and 3,000-cycle SHA2-256 are evident.

		SLH-DSA-SHAKE-*				**SLH-DSA-SHA2-***			
		SLotH		*(PQM4)*		SLotH		*(PQM4)*	
Param.	Func.	clk average	clk/h	*clk/h*	×	clk average	clk/h	*clk/h*	×
128f	KeyGen	176,552	39.3	*13294.6*	*338.5*	358,494	79.8	*3423.4*	*42.9*
	Sign	4,903,978	46.7	*14140.2*	*302.5*	9,127,150	87.0	*3645.8*	*41.9*
	Verify	440,636	71.1	*13405.8*	*188.5*	691,186	111.5	*3413.5*	*30.6*
192f	KeyGen	284,238	43.4	*13500.4*	*310.8*	541,583	82.8	*3461.1*	*41.8*
	Sign	10,596,236	62.7	*14267.0*	*227.4*	23,726,217	140.5	*3786.0*	*26.9*
	Verify	711,431	79.3	*13744.0*	*173.4*	1,290,921	143.9	*3670.8*	*25.5*
256f	KeyGen	815,609	47.5	*13702.4*	*288.7*	1,454,706	84.7	*3480.7*	*41.1*
	Sign	23,660,226	68.5	*14089.4*	*205.6*	50,240,516	145.5	*3710.5*	*25.5*
	Verify	857,059	94.9	*14098.8*	*148.6*	1,419,466	157.1	*3646.5*	*23.2*
128s	KeyGen	11,180,642	38.9	*13294.3*	*342.1*	22,709,640	78.9	*3424.5*	*43.4*
	Sign	102,346,701	46.8	*13306.1*	*284.2*	190,085,952	87.0	*3429.0*	*39.4*
	Verify	179,603	84.5	*13870.8*	*164.2*	268,445	126.2	*3369.9*	*26.7*
192s	KeyGen	18,038,904	43.1	*13497.4*	*313.4*	34,280,105	81.9	*3462.3*	*42.3*
	Sign	263,100,826	69.8	*13492.5*	*193.2*	626,858,593	166.4	*3654.0*	*22.0*
	Verify	289,825	94.7	*13620.7*	*143.8*	641,048	209.5	*3843.6*	*18.4*
256s	KeyGen	13,003,653	47.3	*13691.4*	*289.5*	23,174,830	84.3	*3465.4*	*41.1*
	Sign	296,265,468	90.3	*13674.5*	*151.4*	696,201,400	212.2	*3750.9*	*17.7*
	Verify	469,973	105.6	*13993.7*	*132.5*	894,078	200.9	*3756.7*	*18.7*

PQM4: Cycles for SPHINCS$^+$ simple, accessed Jan 20, 2024: `https://github.com/mupq/pqm4/blob/master/benchmarks.csv`

5.1 Comparison with Other Hardware Accelerators

To our knowledge, SLotH is the first implementation that supports all SLH-DSA parameters and features. The SHA2$^+$ hardware accelerator reported in [29] is of comparable size and application target (OpenTitan Secure Boot), but requires 4.95M cycles for verification with the 256s parameter set. SLotH verification is 10× faster with 0.469M cycles using the SHAKE hash function, and 5× faster at 0.894M cycles using the SHA2 hash functions. Since both designs use a 64-cycle SHA2-256 core, the differences are very likely explained by overhead and underutilization caused by firmware bottlenecks and the lack of hash formatting automation features that SLotH provides.

Early SPHINCS$^+$ FPGA work reported in [2] did not support either SHA2 or SHAKE. The SPHINCS$^+$ implementation reported in [3] is faster than SLotH, but is also larger by a factor of 10, requiring roughly 50,000 Artix-7 LUTs for SHAKE-only parametrizations. Furthermore, it is unclear if it can support more than one parameter set without hardware re-configuration. Such a design is more suitable for an HSM appliance or a network accelerator than for a RoT.

5.2 Comparing SHAKE and SHA2 Variants

The clk/h measure divides the clock count with the total number of high-level hash function invocations in Table 2. These do not correspond exactly to hash-algorithm-dependent core functions, namely Keccak permutations or SHA2 compression function calls. However, using the same hash counts allows us to compare the SHA2 and SHAKE variants with each other. Recall that the Keccak permutation is 24 rounds/cycles while SHA2-256 compression is 64 rounds/cycles. We can observe the rough 64-24 = 40 cycle difference in most clk/h measurements, albeit differing details in hash instantiation (Fig. 1) throw the difference up or down. We note that higher-security variants also require more memory copying (as each hash is n=24 or 32 bytes rather than 16 bytes), which increases the clk/h number for them.

5.3 Comparison to Microcontroller SLH-DSA

Table 5 also includes Cortex M4 clk/h numbers derived from the PQM4 benchmarks [17], which serve to illustrate the performance situation on a security controller without dedicated acceleration. The SHAKE variants are up to 300× faster, while up to 40× acceleration is achieved for SHA2.

It can be easily seen that the roughly 12,000-cycle Keccak permutation and 3,000-cycle SHA2 compression functions completely dominate the clk/h metric for a plain Cortex M4 implementation. In such an implementation there are only small relative gains available from fine-tuning higher-level components or padding steps – but these aspects immediately become obvious bottlenecks when the time required for core hash function computation decreases to 0.2% (SHAKE) or 2% (SHA2) of the original.

5.4 Note on Application-Class Processor Performance

SLotH is a small-area design intended to be integrated with an embedded security controller rather than with a SIMD/Vector-capable main application core, but its cycle counts are significantly better than those reported for corresponding parametrizations of SPHINCS$^+$-simple on x86-64 in [4, Section 10]. There, 56.9M cycles is reported for SPHINCS$^+$-SHAKE-128f-simple signing with AVX2; SLotH performs the same task in 4.9M cycles, despite a vast area difference.

6 Side-Channel and Fault Injection Countermeasures

It is clear that randomized signing makes side-channel attacks much more difficult to mount, so this must be the default in all high-security applications. Timing attacks are not a great concern for SLH-DSA, as conditional branching and memory access patterns are dependent on non-secret authentication paths; they are revealed by the signature and known to a verifier.

As already noted in Sections 2.3 and 2.4, the main target for SLH-DSA leakage countermeasures is the **PRF** function, as it directly handles secret variables.

6.1 Side-Channel Attacks

In [16] Kannwischer, et al. performed a side-channel analysis of an early precursor of SLH-DSA (with a BLAKE hash component). The work describes attacks on its **PRF** component. It is easy to see that the same considerations extend to SLH-DSA. We performed a fixed-vs-random TVLA assessment [7,15] against the master secret SK.seed. Other key components were randomized. Fig. 5 shows leakage from SLH-DSA-SHAKE-128f in an unaccelerated version that computes hashes with the (RISC-V) CPU.

Since such an implementation is very slow, we simply aborted the signing process after 2ms, enough to contain the first **PRF** invocation (which is located in fors_sign/fors_SKgen.) Leakage from SK.seed was rapidly evident, with t value reaching 24.5 in 1,000 traces. The thousands of subsequent **PRF** calls required for each signature would have created similar spikes. We note that leakage from these **PRF** calls can be combined in a horizontal attack as they all use the same SK.seed.

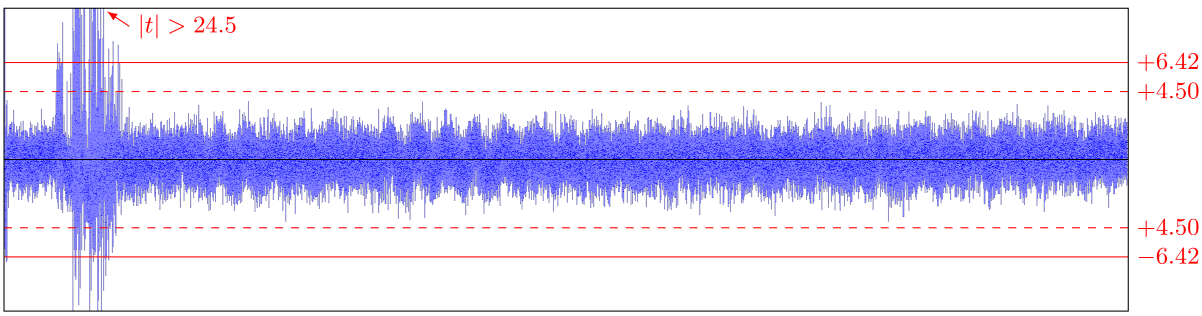

Fig. 5. A TVLA test of a processor implementation of SLH-DSA rapidly shows a leak of SK.seed secret key material. In this t-trace we captured the initial 73k cycles of the signing process, containing the first SHAKE256 **PRF** call. Maximum t value reaches 24.5 already in 1,000 traces. The SHA2-256 **PRF** exhibits similar leakage behavior.

6.2 Masking and Threshold Implementations

In SLH-DSA, some other variables besides the secret key SK itself are also sensitive, although randomization helps to protect them to a degree. Since low members of hash chains (Section 2.4) can be used in forgery attacks, SLotH protects **F** hash chains that follow a **PRF** (these operations are combined and automated.) We do this in WOTS signing and public key generation to protect the entire chain computation. We also mask the "address fixing" **F** that follows **PRF** in FORS signing (line 8 of Alg. 14, fors_node in [26].)

Side-channel protections are somewhat easier to introduce for Keccak than for SHA2, as the relevant techniques such as Threshold Sharing are well-established [9,12]. We observe only a minor performance penalty when using these techniques (Table 6), but the area of the TI Keccak unit is very large (Table 3.)

The current SLotH implementation does not support side-channel countermeasures for SHA2 variants. One large complication in SHA2 masking is caused by the continuous mixing of addition and XOR operations in the algorithm. This requires conversions or Boolean-domain additive arithmetic, which are costly. It appears that masking or other comparable countermeasures will slow down SHA2 significantly more than the SHAKE variants.

We note that the public key component PK.seed is used in virtually every hash, and shows up as false positive leakage in a fixed-vs-random keypair test [28,7,15]. Hence our fixed-vs-random test fixes only the secret key SK.seed and randomizes other components. Recall that three of the four components in the SLH-DSA key (Eq. (2)) are generated randomly, while PK.root is derived from the others. The variable PK.root is not explicitly needed for signature generation.

For positive assurance, we perform a leakage assessment of the SLH-DSA-SHAKE-128f signing function with the TI Keccak module. Fig. 6 shows the result of $N = 100,000$ traces with $L = 5,950,239$ data points each; TVLA hence consists of L independently computed instances of Welch's t-test. Since the number of tests is very large, using the traditional critical value $C = 4.5$ would cause false positives [31]. We adjust the critical value based on trace length L using the Mini-p procedure from Zhang et al. [13], leading to

Table 6. Clock cycles for protected SLotH (TI Keccak) in end-to-end testing (average of 100 iterations.) Verification time is included for completeness (as masking is not used.) The 20-30% overhead comes mainly from setting up and collapsing masked outputs.

Parameter Set	KeyGen		Sign		Verify
SLH-DSA-SHAKE-128f	212,223	+20.2%	5,958,477	+21.5%	440,636
SLH-DSA-SHAKE-192f	354,577	+24.7%	13,789,144	+30.1%	711,431
SLH-DSA-SHAKE-256f	1,004,496	+23.2%	30,395,303	+28.5%	857,059
SLH-DSA-SHAKE-128s	13,456,593	+20.4%	125,533,357	+22.7%	179,603
SLH-DSA-SHAKE-192s	22,530,331	+24.9%	348,715,357	+32.5%	289,825
SLH-DSA-SHAKE-256s	16,022,620	+23.2%	391,246,520	+32.1%	469,973

$C = 7.06$. The maximum spike in testing was $|t| = 5.00$. As with other tests in this paper, we used the XC7A100T FPGA chip of a CW305 board, and measured power traces from the board with a PicoScope 2208B oscilloscope. The sampling frequency was the same as the clock frequency of the target FPGA, 31.25 MHz.

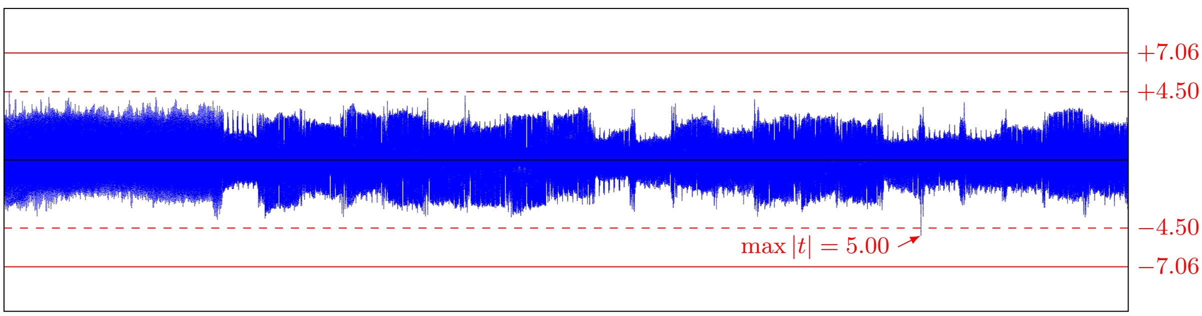

Fig. 6. 100,000 traces of the core signing process (5.95M cycles, one sample per cycle at 31.24MHz) has a maximum spike of $|t| = 5.00$, which is below the long-trace adjusted critical value $C = 7.06$.

6.3 Custom PRF Hardening Options

The thousands of "secret key expansion" **PRF** calls are modeled as random bits in SLH-DSA security proofs and as long as **PRF** produces deterministic outputs at given addresses (ADRS), SLH-DSA works. Matching **PRF** output values need to be used for public key generation too, of course, but the verification process will not require any modification if **PRF** is changed. Since signature and public key formats are unmodified, a signing module hardened this way is externally interoperable and can be used transparently in most SLH-DSA applications.

Hence, if deviance from the specification of the signing function is allowed, one can consider using a secure Keccak-based **PRF** implementation with otherwise SHA2-based SLH-DSA or replacing the **PRF** function with something completely different. For example, using different secret seed values at different parts of the algorithm reduces the exposure in attack compared to directly using the same master value SK.seed in thousands of hashes [6]. One can also consider using a custom-designed side-channel secure **PRF** component.

6.4 Practical Security Considerations

We note that SLotH offers a substantial security improvement over all CPU-based implementations even if masking is not used. This is because the master secret key SK.seed is held in a hardware register: `KECC_SKSD` for Keccak (Table 7) and `S256_SKSD` for SHA2 (Table 8). The Threshold Implementation further splits it into three Boolean shares (SK.seed = `KTI3_SKSA` ⊕ `KTI3_SKSB` ⊕ `KTI3_SKSC`) that are continuously refreshed (Table 7).

Even without masking, the firmware component can forget SK.seed after it has been loaded into a SLotH hash unit once. The hardware always uses it as a whole, loading it into the respective hash engines in a single cycle for **PRF** computation, where it is very rapidly processed. Hence an attacker gains significantly less information than from a CPU implementation that spends many cycles processing each key word separately, resulting in leakage as shown in Fig. 5. In such a single-cycle hardware register load, the main dynamic power "toggling" comes for the state change of the 1600-bit Keccak register or 512-bit SHA2-256 message block; essentially the Hamming distance between the final state of the previous hash computation and the new **PRF** input message block. This significantly complicates key recovery attacks in practice.

Hence from a practical viewpoint, even unmasked SLotH gives substantial side-channel protection due to its secret key management, very wide data paths, and fast speed. Fig. 7 contains a TVLA evidence of $N = 10,000$ traces with the unmasked Keccak module, without the "plaintext key load" initialization step.

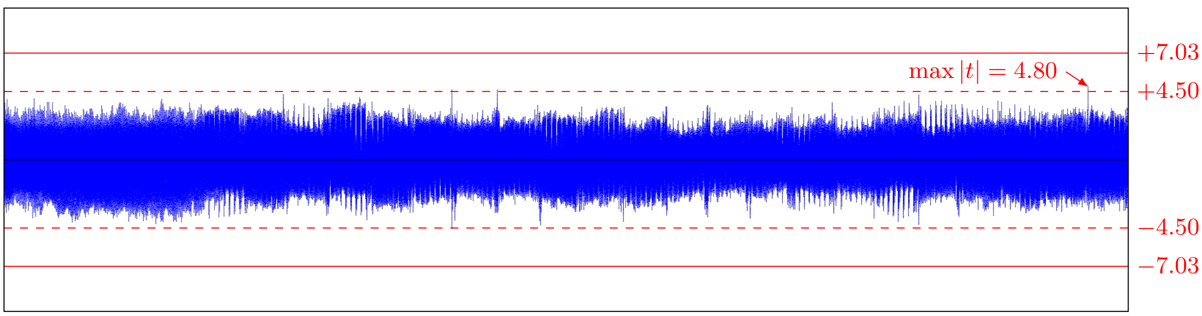

Fig. 7. 10,000 traces of the core signing process (SLH-DSA-SHAKE-128f) has a maximum spike at $|t| = 4.80$, which is below the critical value $C = 7.03$ with 4.91M time points . The secret keys are entirely handled by hardware, reducing their exposure to side-channel attacks.

6.5 Redundancy for Fault Injection Protection

As discussed in [10,3] and further emphasized by recent work by Genêt and others [14,30], SLH-DSA is surprisingly fragile against Fault Injection Attacks (FIA). Since such attacks are highly relevant for the RoT applications where SLotH is intended, redundancy is required. The relatively small size of SLotH allows one to instantiate two or more copies of it in hardware. To counter targeted faults and dual-fault attacks, each instance should be independently randomized, and special care must be taken when implementing cross-checks. We leave this for future work.

7 Conclusions

We have described SLotH, a new open-source accelerator designed specifically to support FIPS 205 SLH-DSA [26] in SoC Root-of-Trust (RoT) systems. SLotH supports all 12 parameter sets of the new standard and offers approximately two orders of magnitude faster performance when compared to a RoT without acceleration. We observe that one can make SLH-DSA 10× faster simply by making a fast hardware hash function available [29], but to make it 100× faster, one needs to design and optimize the software-hardware control specifically for SLH-DSA. However, this does not greatly increase the hardware area.

With these optimizations, SLH-DSA signature verification (required for RoT-supported secure boot) is arguably faster and more energy efficient than the corresponding functionality provided by ML-DSA [24] or ECDSA [23] accelerators of comparable size. For example, the OpenTitan Big Number (OTBN) accelerator[¶]

[¶] OpenTitan Big Number performance: `https://opentitan.org/book/hw/ip/otbn/doc/otbn_intro.html` performs ECDSA P-256 signature verification in 420,220 cycles, while SLH-DSA-SHAKE-128s verification with SLotH is 179,603 cycles. Similarly, 1,075,092 cycles are reported for ECDSA-P384 verification with OTBN, while SLH-DSA-SHAKE-192s verification with SLotH is 289,825 cycles.

The SLotH control firmware has been newly developed to reduce memory copying and other CPU bottlenecks that become apparent when core hash function computation no longer dominates the overall cycle count. SLotH also has reduced firmware/ROM size as the same high-level (portable) algorithm code is shared by all variants and parameters are not hard-coded into macros. Firmware that supports all parameter sets fits into 17 kB, while 64 kB of working RAM is sufficient to run the algorithms and hold SLH-DSA signatures (that can be up to 50 kB in size). The co-design remains relatively compact in terms of hardware area, containing only one instance of each hash compression function; the speed-up compared to prior works is achieved mainly by optimizing their utilization. Category 1 SLH-DSA-SHA2 unit (with SHA2-256 only) is the smallest configuration (32.03 kGE in addition to the CPU), but SHA2-512 support required for high-security (Category 3 and 5) SLH-DSA-SHA2 increases the area requirement to 82.36 kGE. The high-performance Keccak unit supports all parameter sets with an area requirement of 41.72 kGE.

We also offered an analysis of the sensitivity of SLH-DSA signing against power and electromagnetic side-channel attacks and suitable countermeasures to protect against them. We first demonstrated the vulnerability of CPU-based SLH-DSA implementations by showing SK.seed secret key leakage from the **PRF** function. We then perform a TVLA leakage assessment of our protected implementation with $N = 100,000$ traces of the full signing function. We further consider the practical security increase obtained by hardware key management of SLotH even when masking is not used, and special design ideas such as a "custom **PRF**" which still has interoperable signature verification.

We note that SLH-DSA has an especially strong requirement for redundancy and careful consistency checking due to the fragility of the scheme against fault injection attacks [14,30]. We are working to create functionally duplicated, redundant instances of SLotH for this purpose, which requires special considerations. However, the self-contained nature and compact size of SLotH makes this a feasible option.

Hardware and firmware source code of SLotH is available from: `https://github.com/slh-dsa/sloth`

References

1. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status report on the third round of the NIST post-quantum cryptography standardization process. Interagency or Internal Report NISTIR 8413-upd1, National Institute of Standards and Technology (September 2022). https://doi.org/10.6028/NIST.IR.8413-upd1
2. Amiet, D., Curiger, A., Zbinden, P.: FPGA-based accelerator for post-quantum signature scheme SPHINCS-256. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 18–39 (2018). https://doi.org/10.13154/TCHES.V2018.I1.18-39
3. Amiet, D., Leuenberger, L., Curiger, A., Zbinden, P.: FPGA-based SPHINCS$^+$ implementations: Mind the glitch. In: 23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020. pp. 229–237. IEEE (2020). https://doi.org/10.1109/DSD51259.2020.00046
4. Aumasson, J.P., Bernstein, D.J., Beullens, W., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Westerbaan, B.: SPHINCS+ – submission to the 3rd round of the NIST post-quantum project. v3.1 (June 2022), https://sphincs.org/data/sphincs+-r3.1-specification.pdf
5. Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R., Simon, S.: Recommendation for pair-wise key establishment using integer factorization cryptography. NIST Special Publication SP 800-56B Rev 2 (March 2019). https://doi.org/10.6028/NIST.SP.800-56Br2
6. Bashiri, K.: Personal communication. BSI, Bonn (January 2024)
7. Becker, G., Cooper, J., DeMulder, E., Goodwill, G., Jaffe, J., Kenworthy, G., Kouzminov, T., Leiserson, A., Marson, M., Rohatgi, P., Saab, S.: Test vector leakage assessment (TVLA) methodology in practice (2013), presented at International Cryptography Module Conference – ICMC 2013
8. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS$^+$ signature framework. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 2129–2146. ACM (2019). https://doi.org/10.1145/3319535.3363229, https://eprint.iacr.org/2019/1086, full version is available as IACR ePrint Report 2019/1086

9. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Building power analysis resistant implementations of Keccak (August 2010), https://csrc.nist.gov/Events/2010/The-Second-SHA-3-Candidate-Conference
10. Castelnovi, L., Martinelli, A., Prest, T.: Grafting trees: A fault attack against the SPHINCS framework. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10786, pp. 165–184. Springer (2018). https://doi.org/10.1007/978-3-319-79063-3_8, https://eprint.iacr.org/2018/102, full version is available as IACR ePrint Report 2018/102
11. Cooper, D.A., Apon, D.C., Dang, Q.H., Davidson, M.S., Dworkin, M.J., Miller, C.A.: Recommendation for stateful hash-based signature schemes. NIST Special Publication SP 800-208 (October 2020). https://doi.org/10.6028/NIST.SP.800-208
12. Daemen, J.: Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10529, pp. 137–153. Springer (2017). https://doi.org/10.1007/978-3-319-66787-4_7
13. Ding, A.A., Zhang, L., Durvaux, F., Standaert, F., Fei, Y.: Towards sound and optimal leakage detection procedure. In: Eisenbarth, T., Teglia, Y. (eds.) Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10728, pp. 105–122. Springer (2017). https://doi.org/10.1007/978-3-319-75208-2_7
14. Genêt, A.: On protecting SPHINCS+ against fault attacks. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2023**(2), 80–114 (2023). https://doi.org/10.46586/TCHES.V2023.I2.80-114
15. ISO: Information technology – security techniques – testing methods for the mitigation of non-invasive attack classes against cryptographic modules. Draft International Standard ISO/IEC DIS 17825:2022(E), International Organization for Standardization (2023)
16. Kannwischer, M.J., Genêt, A., Butin, D., Krämer, J., Buchmann, J.: Differential power analysis of XMSS and SPHINCS. In: Fan, J., Gierlichs, B. (eds.) Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10815, pp. 168–188. Springer (2018). https://doi.org/10.1007/978-3-319-89641-0_10, https://eprint.iacr.org/2018/673
17. Kannwischer, M.J., Petri, R., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4 (2024), https://github.com/mupq/pqm4
18. McGrew, D., Curcio, M., Fluhrer, S.: Leighton-Micali Hash-Based Signatures. RFC 8554 (April 2019). https://doi.org/10.17487/RFC8554
19. Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of nonlinear functions in the presence of glitches. J. Cryptol. **24**(2), 292–321 (2011). https://doi.org/10.1007/s00145-010-9085-7
20. NIST: The keyed-hash message authentication code (HMAC). Federal Information Processing Standards Publication FIPS 198-1 (July 2008). https://doi.org/10.6028/NIST.FIPS.198-1
21. NIST: Secure hash standard (SHS). Federal Information Processing Standards Publication FIPS 180-4 (August 2015). https://doi.org/10.6028/NIST.FIPS.180-4
22. NIST: SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication FIPS 202 (August 2015). https://doi.org/10.6028/NIST.FIPS.202
23. NIST: Digital signature standard (DSS). Federal Information Processing Standards Publication FIPS 186-5 (February 2023). https://doi.org/10.6028/NIST.FIPS.186-5
24. NIST: Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standards Publication FIPS 204 (Draft) (August 2023). https://doi.org/10.6028/NIST.FIPS.204.ipd
25. NIST: Module-Lattice-based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication FIPS 203 (Draft) (August 2023). https://doi.org/10.6028/NIST.FIPS.203.ipd
26. NIST: Stateless Hash-Based Digital Signature Standard. Federal Information Processing Standards Publication FIPS 205 (Draft) (August 2023). https://doi.org/10.6028/NIST.FIPS.205.ipd
27. NSA: The commercial national security algorithm suite 2.0 and quantum computing FAQ. National Security Agency, Cybersecurity Information Sheet (September 2022), https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/0/CSI_CNSA_2.0_FAQ_.PDF
28. Schneider, T., Moradi, A.: Leakage assessment methodology - A clear roadmap for side-channel evaluations. In: Güneysu, T., Handschuh, H. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9293, pp. 495–513. Springer (2015). https://doi.org/10.1007/978-3-662-48324-4_25
29. Wagner, A., Oberhansl, F., Schink, M.: To be, or not to be stateful: Post-quantum secure boot using hash-based signatures. In: Chang, C., Rührmair, U., Mukhopadhyay, D., Forte, D. (eds.) Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security, ASHES 2022, Los Angeles, CA, USA, 11 November 2022. pp.

85–94. ACM (2022). https://doi.org/10.1145/3560834.3563831, https://eprint.iacr.org/2022/1198, also available as IACR ePrint Report 2022/1198
30. Wagner, A., Wesselkamp, V., Oberhansl, F., Schink, M., Strieder, E.: Faulting Winternitz one-time signatures to forge LMS, XMSS, or SPHINCS$^+$ signatures. In: Johansson, T., Smith-Tone, D. (eds.) Post-Quantum Cryptography - 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14154, pp. 658–687. Springer (2023). https://doi.org/10.1007/978-3-031-40003-2_24, https://eprint.iacr.org/2023/1572, also available as IACR ePrint Report 2023/1572
31. Whitnall, C., Oswald, E.: A critical analysis of ISO 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11923, pp. 256–284. Springer (2019). https://doi.org/10.1007/978-3-030-34618-8_9

Table 7. Register map for SLotH's unmasked and masked Keccak units (Section 3.1).

Register Name	**Offset**	**Brief description**
KECCAK_BASE_ADDR	(0)	Base address, in prototype 0x15000000.
KECC_MEMA	0x0000	1600-bit Keccak permutation input-output state A.
KECC_ADRS	0x00c8	32-byte ADRS structure for hash formatting.
KECC_SEED	0x00e8	Public key variable PK.seed for hash formatting.
KECC_SKSD	0x0108	Secret key SK.seed for **PRF** computation.
KECC_MTOP	0x0128	*End of data / state register space.*
KECC_CTRL	0x01e0	Control and status: Write 0x01 to start raw Keccak f1600, read for status (0x00=ready).
KECC_STOP	0x01e4	Round count (for TurboShake / KangarooTwelve).
KECC_SECN	0x01e8	Security / field length write $n \in \{16, 24, 32\}$.
KECC_CHNS	0x01ec	Iteration count & trigger for hash chaining. Set to s for s iterations. Set to 0x40 + s for **PRF** + hashes. Set to 0x80 to perform initial padding for **H** or $\mathbf{T}_\ell$.

Register Name	**Offset**	**Brief description**
KECTI3_BASE_ADDR	(0)	Base address, in prototype 0x14000000.
KTI3_MEMA	0x0000	1600-bit Keccak permutation input-output state A.
KTI3_MEMB	0x00c8	Keccak secret state share B.
KTI3_MEMC	0x0190	Keccak secret state share C.
KTI3_ADRS	0x0260	32-byte ADRS structure for hash formatting.
KTI3_SEED	0x0280	Public key variable PK.seed for hash formatting.
KTI3_SKSA	0x02a0	Secret key SK.seed for **PRF**, Share A.
KTI3_SKSB	0x02c0	Secret key SK.seed share B.
KTI3_SKSC	0x02e0	Secret key SK.seed share C.
KTI3_MTOP	0x0300	*End of data / state register space.*
KTI3_CTRL	0x03c0	Control and status: Write 0x01 to start raw Keccak f1600, read for status (0x00=ready).
KTI3_STOP	0x03c4	Round count (for TurboShake / KangarooTwelve).
KTI3_SECN	0x03c8	Security / field length write $n \in \{16, 24, 32\}$.
KTI3_CHNS	0x03cc	Iteration count & trigger for hash chaining. Set to s for s iterations. Set to 0x40 + s for **PRF** + hashes. Set to 0x80 to perform initial padding for **H** or $\mathbf{T}_\ell$.

Table 8. Control registers for SLotH SHA2-256 and SHA2-512 units (Section 3.2).

Register Name	Offset	Brief description
SHA256_BASE_ADDR	(0)	Base address, in prototype at 0x16000000.
S256_HASH	0x0000	32-byte hash chaining variable (IV/output).
S256_MSGB	0x0020	64-byte message block input.
S256_SEED	0x0060	Precomputed result of processing a zero-padded PK.seed block (for formatting).
S256_ADRS	0x0080	32-byte ADRS structure. Automatically compressed into 22-byte ADRS^c for message formatting.
S256_SKSD	0x00A0	Secret key SK.seed for **PRF** computation.
S256_MTOP	0x00C0	*End of main data/register state.*
S256_MSH2	0x0100	Start of 2-byte offset alignment mirror.
S256_CTRL	0x01e0	Control: Write for trigger (0x01 to start Keccak, 0x02 to start chain iteration, 0x03 for **PRF** + chain.)
S256_SECN	0x01e8	Security / field length parameter $n \in \{16, 24, 32\}$.
S256_CHNS	0x01ec	Iteration count for Winternitz chain (not a trigger.) Set to 0x00 to only perform state formatting.

Register Name	Offset	Brief description
SHA512_BASE_ADDR	(0)	Base address, in prototype at 0x17000000.
S512_HASH	0x0000	64-byte hash chaining variable (IV/output).
S512_MSGB	0x0040	128-byte message block input.
S512_MTOP	0x00c0	*End of main data/register state.*
S512_MSH2	0x0100	Start of 2-byte offset alignment mirror.
S512_CTRL	0x01e0	Control: Write 0x01 to trigger the compression function. Read for status (0x00 = done.)