

“SLOTH”

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

April 11, 2024 – 5th NIST PQC Standardization Conference

Markku-Juhani O. Saarinen

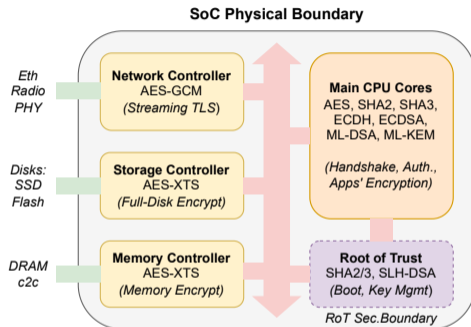
markku-juhani.saarinen@tuni.fi

NISEC and SoC Hub Research Center, Tampere University

SlotH: SLH-DSA Architecture for SoC Root-of-Trust (RoT) Units

SlotH Features:

- ▶ Keccak (SHAKE) and SHA2 (256/512) hardware: Supports all parameter sets of SLH-DSA in FIPS 205 ipd.
- ▶ Not much larger than existing Root-of-Trust hash accelerators.
- ▶ But often 10 times faster due to SLH-DSA specific optimizations.
- ▶ Plus: An Experimental 3-share TI Keccak for side-channel security.



SLH-DSA is used by RoTs to verify system integrity: updates, boot signatures.

Full software and hardware source code: <https://github.com/slh-dsa/sloth>

10 × Faster than big CPUs, over 100 × Faster than Embedded SW

Example: SLH-DSA-SHAKE-128f (SPHINCS⁺-SHAKE-128f-simple) cycle counts.

Implementation	KeyGen	Sign	Verify
pqm4: Embedded SW [1]	59,759,081	1,483,676,214	83,065,165
avx2: Main CPU SW [2]	2,249,444	56,933,788	3,346,068
shake256_lsu HW [3]	1,724,534	42,597,665	2,457,742
SLotH [this work]	176,552	4,903,978	440,636
Gain over embed SW [1]	338.5×	302.5×	188.5×
Gain over AVX2 SW [2]	12.7×	11.6×	7.6×
Gain over LSU HW [3]	9.8×	8.7×	5.6×

- [1] M. J. Kannwischer, R. Petri, J. Rijneveld, P. Schwabe, K. Stoffelen: “PQM4: Post-quantum crypto library for the ARM Cortex-M4.” (Talk here too), 2024. <https://github.com/mupq/pqm4>
- [2] SPHINCS⁺ Team: “SPHINCS⁺ Submission to the NIST post-quantum project, v.3.1.” June 2022. <https://sphincs.org/data/sphincs+-r3.1-specification.pdf>
- [3] P. Karl, J. Schupp, G. Sigl: “The Impact of Hash Primitives and Communication Overhead for Hardware-Accelerated SPHINCS+,” COSADE 2024 (April 9–10), 2024. <https://ia.cr/2023/1767>

.. Why/How?

Why is current software and hardware so much worse?

- ▶ Hashes are very fast in hardware, and very slow on CPUs:
SHA3/SHAKE (Keccak f1600) is 24 cycles in HW, 2,000 (..10,000+) cycles on CPUs.
SHA2 (256/512 compr. func) is 64/80 cycles in HW, 1,000+ cycles on CPUs.
- ▶ Hash accelerators are designed to hash data, not to hash other hashes.
- ▶ Time is wasted while the CPU setting up new data to be hashed.

How did we make it faster: *(Perform quantitative analysis, remove bottlenecks.)*

- ▶ Automate hash primitive formats in hardware, minimizing CPU involvement:
Hold keys (PK.seed and SK.seed) and ADRS fields in hardware registers.
- ▶ Automate Winternitz iteration – most of SLH-DSA is performing iteration (2):

$$X^0 = \text{PRF}(\text{PK.seed}, \text{SK.seed}, \text{ADRS} = \text{WOTS_PRF}) \quad (1)$$

$$X^j = F(\text{PK.seed}, \text{ADRS} = \text{WOTS_HASH}(j), X^{j-1}) \text{ for } j \geq 1. \quad (2)$$

SLH-DSA Hash Primitive Formats (1/2)

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

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

$\underline{\text{PRF}_{\text{msg}}(\text{SK.prf}, \text{opt_rand}, M)}$ (PQ-PRF) Used in:
= SHAKE256($\text{SK.prf} \parallel \text{opt_rand} \parallel M, 8n$) *SHAKE, all*
= $\text{Trunc}_n(\text{HMAC-SHA-256}(\text{SK.prf}, \text{opt_rand} \parallel M))$ *SHA2, $n = 16$*
= $\text{Trunc}_n(\text{HMAC-SHA-512}(\text{SK.prf}, \text{opt_rand} \parallel M))$ *SHA2, $n \geq 24$*

SLH-DSA Hash Primitive Formats (2/2)

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

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

$\underline{T_\ell(\text{PK.seed}, \text{ADRS}, M_\ell)}$ (PQ-DM-SPR) Used in:
= SHAKE256(PK.seed || ADRS || M_ℓ , $8n$) *SHAKE, all*
= Trunc _{n} (SHA-256(PK.seed || toByte(0, $64 - n$) || ADRS^c || M_ℓ)) *SHA2, $n = 16$*
= Trunc _{n} (SHA-512(PK.seed || toByte(0, $128 - n$) || ADRS^c || M_ℓ)) *SHA2, $n \geq 24$*

Hash Primitive Counts: `slh_sign()`, Signature Generation

Distribution of hash primitive calls in SLH-DSA-SHA2- and SLH-DSA-SHAKE-* signing.*

Function	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
T_ℓ	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%

- ▶ A large majority of signing work is in F calls in `chain()` – Winternitz iteration.
- ▶ Perhaps 10% of calls are PRF calls that use the secret key `SK.seed`.
- ▶ (This table excludes H_{msg} and PRF_{msg} as those are called only 1 or 2 times.)

Hash Primitive Counts: `slh_verify()`, Signature Verification

Distribution of hash primitive calls in SLH-DSA-SHA2- and SLH-DSA-SHAKE-* verification.*

Function	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
T_ℓ	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%

- ▶ More than 90% of verification work is in F calls in chain() – Winternitz iteration.
- ▶ The “small” parameter sets (**s**) require *fewer* hashes than the “fast” parameter sets (**f**). For *verification*, **s** parameter signatures are actually much faster.

On SLoTH Hardware and Firmware

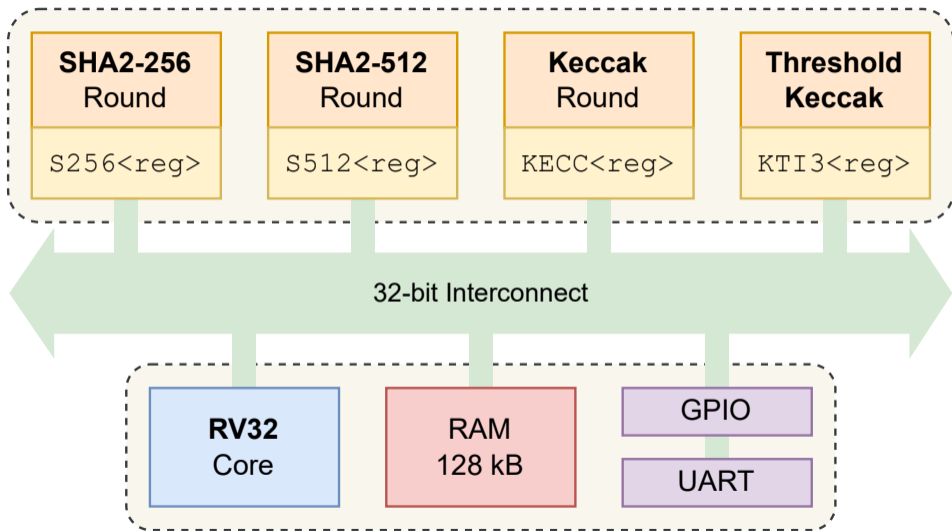
Full hardware and software for the prototype:

<https://github.com/slh-dsa/sloth>

Some Features and Notes:

- ▶ About 6,700 lines of bare metal ANSI C and 4,100 lines of Verilog.
- ▶ I wrote it mostly from scratch after the publication of FIPS 205 ipd.
- ▶ Actually Free: BSD 3-Clause License, no patent applications, etc.
- ▶ Shared implementation for all 12 parameters; 16.4kB binary “ROM” for all.
- ▶ Works with 64kB RAM (4kB stack – recall that signatures are up to 50kB).
- ▶ Clean split between the “algorithm core” and “hardware driver” components. Software part “slh” also runs without special hardware (on any PC).
- ▶ Known Answer Tests (KAT) match with post-FIPS updated Reference code.

Block Diagram: Straightforward Memory-Mapped units (no DMA)



Example Register Map: KTI3_<reg> Threshold Keccak

Register Name	Offset	Bytes	Brief description
_BASE_ADDR	(0)	(1024)	Memory-mapped in prototype at 0x14000000.
KTI3_MEMA	0x0000	200	1600-bit Keccak permutation input-output state A.
KTI3_MEMB	0x00c8	200	Keccak secret state share B. <i>(Only in T13.)</i>
KTI3_MEMC	0x0190	200	Keccak secret state share C. <i>(Only in T13.)</i>
KTI3_ADRS	0x0260	32	32-byte ADRS structure for hash formatting.
KTI3_SEED	0x0280	32	Public key variable PK.seed for hash formatting.
KTI3_SKSA	0x02a0	32	Secret key SK.seed for PRF , share A.
KTI3_SKSB	0x02c0	32	Secret key SK.seed for PRF , share B. <i>(Only in T13.)</i>
KTI3_SKSC	0x02e0	32	Secret key SK.seed for PRF , share C. <i>(Only in T13.)</i>
KTI3_CTRL	0x03c0	4	Raw function control and status: Write 0x01 to start raw Keccak f1600, read for status (0x00=ready).
KTI3_STOP	0x03c4	4	Round count (for TurboShake / KangarooTwelve).
KTI3_SECN	0x03c8	4	Security / field length write $n \in \{16, 24, 32\}$.
KTI3_CHNS	0x03cc	4	Iteration count & trigger for hashing and chaining. <ul style="list-style-type: none">- Set to s for s Winternitz F iterations.- Set to $0x40 + s$ for PRF + s Winternitz F iterations.- Set to $0x80$ to perform initial padding for H or T_ℓ.

Configurable Hardware: Artix 7 FPGA LUTs / ASIC Gate Equivalents

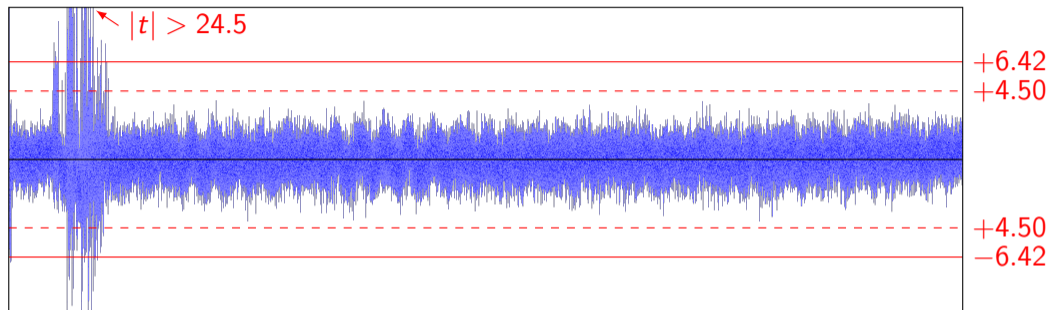
CPU+IX RV32IMC	Keccak "plain"	SHA2 -256	SHA2 -512	Keccak TI3	LUTs XC7A100T	kGE Nangate45
yes	-	-	-	-	(3,023)	(31.36)
yes	-	yes	-	-	+2,463	+32.03
yes	yes	-	-	-	+5,582	+41.72
yes	yes	yes	-	-	+8,205	+73.52
yes	-	yes	yes	-	+5,942	+82.36
yes	yes	yes	yes	-	+10,857	+123.99
Full system, all SLH-DSA parameters:					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

Side Channels: Sensitive Variable Leakage

- ▶ SLH-DSA's **master secret** is **SK.seed** (with randomization **SK.prf** is redundant.) Also: Many of the hashes are *ephemeral* secrets – allowing forgeries, if leaked.
- ▶ SlotH has a simple countermeasure of masked (TI) PRF + Winternitz chaining. Note: The **PRF** key expander can be modeled as a random function of **ADRS**. One can use a “*custom PRF*” without breaking interoperability with verification.
- ▶ A major issue for SLH-DSA in a RoT are **fault attacks**. Genêt [1] shows that: A random bit-flip fault during signing can cause signatures to be generated that *will verify as correct* while containing hashes that allow *universal forgeries*. SlotH is relatively small & flexible; we can add more redundancy (future work.)

[1] Aymeric Genêt: “On Protecting SPHINCS+ Against Fault Attacks.”, CHES/TCHES 02/2023, <https://ia.cr/2023/042>, 2023.

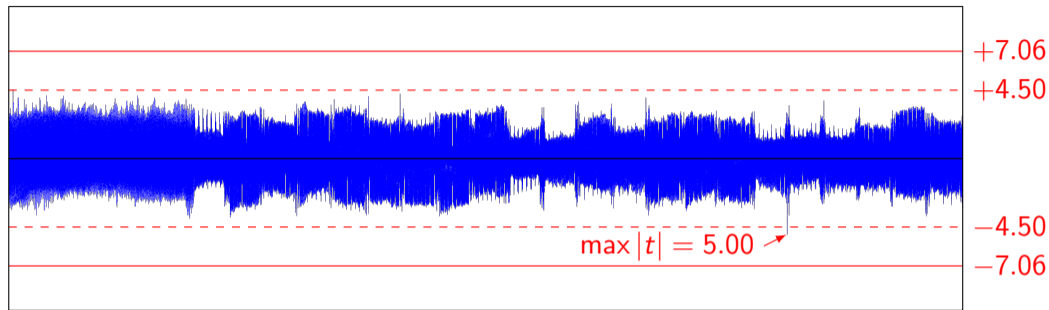
Unprotected CPU Implementations Leak **SK.seed**



Zoom of the first PRF in a non-accelerated TVLA shows strong leakage.

- ▶ Each SLH-DSA Signing operation has thousands of invocations of PRF, each using SK.seed. So even a 1-trace (horizontal) attack reveals secret key bits.
- ▶ Implementation is very slow; just demonstrating leakage from the first PRF.

Positive Assurance: N=100,000 Traces of SLoth with TI3



SK.seed autoloading + TI3 Keccak. TVLA: $N = 100\ 000$, $L = 5\ 950\ 239$, $C = 7.06$.

- ▶ TVLA with 3-share TI Keccak for PRF (*SK.seed*) and secret Winternitz hashes.
- ▶ Countermeasure doubles hardware size, but less than 25% performance hit.
- ▶ Even without TI3 Keccak, this implementation is reasonably secure due to its parallel (1-cycle) loading of secrets. The software can “forget” secret key!

Performance (1/2): "Fast signature" (f) parameter sets

	SLH-DSA-SHAKE-*				SLH-DSA-SHA2-*			
	SLoth		(PQM4)		SLoth		(PQM4)	
Func.	clk average	clk/h	<i>clk/h</i>	×	clk average	clk/h	<i>clk/h</i>	×
128f KG	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 KG	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 KG	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

- ▶ SLH-DSA-SHAKE-128f signing is 4.9M cycles or 19.6ms @ 250 MHz (XCVU9P).
- ▶ SHA2 variants are about half the speed of SHAKE (it's a slower hash in HW.)
- ▶ SHAKE is 150-300× faster than embedded SW, SHA2 about 25-40× faster.

Performance (2/2): "Small signature" (s) parameter sets

Func.	SLH-DSA-SHAKE-*				SLH-DSA-SHA2-*			
	SLotH	(PQM4)			SLotH	(PQM4)		
	clk average	clk/h	<i>clk/h</i>	×	clk average	clk/h	<i>clk/h</i>	×
128s KG	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 KG	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 KG	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

- ▶ SLH-DSA-SHAKE-128s verification is only 179.6k cycles or 0.72ms @ 250 MHz.
- ▶ But signing with s variants is of course 20× slower than with f variants.
- ▶ Core hash utilization even with SHAKE is often within 50% of optimal.

Final Notes and Conclusions

- ▶ SLOtH is a free, fully open-source SLH-DSA accelerator architecture under development (for SoC RoTs). <https://github.com/slh-dsa/sloth>

Findings:

- ▶ You can make SLH-DSA about 10× faster on hash accelerators by automating message formats (PK.seed, SK.seed, ADRS registers) and Winternitz chain().
Useful reminder: Quantitative analysis is essential for understanding bottlenecks.

Side-Channel Security:

- ▶ Having a hardware SK.seed register, fast/parallel hash set-up helps a lot.
- ▶ SLOtH has a 3-share TI Keccak option – very big, but fully KAT compatible.
- ▶ Custom PRF's can be considered – verification remains compatible.
- ▶ However, fault attacks remain a big problem for SLH-DSA [Genêt, CHES 2023].