FPGA-based SPHINCS$^+$ Implementations: Mind the Glitch

Dorian Amiet*, Lukas Leuenberger*, Andreas Curiger† and Paul Zbinden*
* IMES Institute for Microelectronics and Embedded Systems, OST Eastern Switzerland University of Applied Sciences, Switzerland
Email: dorian.amiet@ost.ch, lukas.leuenberger@ost.ch, paul.zbinden@ost.ch
† Securosys SA, Zürich, Switzerland
Email: curiger@securosys.ch

***Abstract*—The digital signature scheme SPHINCS$^+$ is a candidate in the NIST post-quantum project, whose aim is to standardize cryptographic systems that are secure against attacks originating from both quantum and classical computers. We present an efficient and, to our knowledge, first hardware implementation for SPHINCS$^+$. Our systematic approach of a performance-optimized FPGA architecture results in a 100x speed-up compared to the reference software-only implementation. Our investigation on a real-world implementation revealed a weakness regarding fault injection. The attack breaks the scheme completely. Collecting enough private information to forge a signature is a matter of seconds on our setup. We discuss possible countermeasures. A "sign-then-verify" operation unfortunately does not detect a faulty signature, but a full replication of the hardware might make a detection possible.**

***Index Terms*—SPHINCS$^+$, post-quantum cryptography, FPGA, fault attack**

I. Introduction

Large-scale quantum computers might be able one day to break today's signature schemes in use (RSA [1] and ECDSA [2]) by running Shor's algorithm [3]. In 2017, NIST started a standardization process [4] for post-quantum algorithms, i.e. cryptographic algorithms able to withstand attacks that would benefit from the processing power of quantum computers. Proposed algorithms in this process include digital signature schemes, key exchange mechanisms and asymmetric encryption. In 2019, 26 of the primary 69 candidates were selected to move to the second round [5]. This second round includes 9 signature schemes now, in which only one scheme is following a hash-based approach, namely SPHINCS$^+$ [6], which was submitted by Andreas Hülsing et al.

SPHINCS$^+$ describes a general signature scheme which can be combined with an arbitrary hash function. Several parameters allow trade-offs between computational effort, signature size, and security margin. The authors provide six different parameter sets for security and performance tradeoffs, a 'simple' and 'robust' option, and a selection of three hash functions. This leads to 36 different instances of SPHINCS$^+$. All of them impress with small key sizes and moderate signature sizes of 8 to 50 kbytes. Hash-based signature schemes can usually be proved to be secure as long as the underlying hash function is considered secure [7]. Therefore, hash-based signature schemes seem to be a reasonable choice regarding security compared to other post-quantum signature schemes. Moreover, in contrast to most other hash-based signature schemes, SPHINCS$^+$ is a stateless signature scheme, which makes it a promising replacement candidate for today's widely used signature schemes.

A sign of weakness of the SPHINCS$^+$ scheme seems to be its signing time. The SPHINCS$^+$ authors report latencies ranging from 6.5 milliseconds (partly hardware accelerated) up to a few seconds on a 3.5 GHz processor [6]. To increase the throughput of cryptographic algorithms, it is common practice to use specialized co-processors. To our knowledge, SPHINCS-256 [8], the predecessor of SPHINCS$^+$, is the only stateless hash-based signature scheme on which a hardware-based accelerator has been reported [9]. A different approach to speedup SPHINCS-256 signing is to use powerful Graphics Processing Units and calculate many signatures in parallel [10]. Another publication [11] presents a co-processor for the stateful hash-based signature scheme XMSS [12].

All the above-mentioned publications lack a security analysis regarding implementation-specific attacks. Active side-channel attacks (also referred to as fault attacks) try to alter the processing of an algorithm. Boneh et al. published one of the first descriptions of such an attack in 1997 [13]. The goal is often to either skip a branch or to change the computation of an algorithm in a way to make it leak private data. A good overview of different active side-channel attacks can be found in [14] and [15], respectively.

The contribution of this paper includes the (according to our knowledge) first hardware-based implementation of SPHINCS$^+$ including performance results for all SHAKE256-based variants. In addition, we provide a fault attack on the proposed hardware architecture and a discussion on countermeasures. We point out that countermeasures within the algorithm (without special fault detection hardware) might be quite inefficient.

The paper is structured as follows: In Section II, the SPHINCS$^+$ scheme is recalled. Our SPHINCS$^+$ FPGA architecture is described in Section III. Details on its performance are provided in Section IV. Deep analysis on a successful active attack on our SPHINCS$^+$ implementation is presented in Section V.

II. Hash-based Signatures

Using cryptographic hash functions (digest = $f(\text{seed})$) for signing is not a new idea at all. As early as 1979, L. Lamport, R. C. Merkle, and R. Winternitz introduced hash-based signature schemes ([16] and [17]). Several improvements to these basic schemes led to the SPHINCS$^+$ scheme, which we describe in this section.

A. Winternitz One-Time Signature WOTS

A WOTS key pair can be used to generate exactly one signature. It allows for a trade-off between signature size and processing effort by the Winternitz parameter w. The WOTS construction has the advantage that the public key can be derived from a message and its signature.

An n-bit long message m is split into $n/log_2(w)$ sub-message chunks m_i. Each chunk is interpreted as an unsigned number $0 \leq m_i < w$. For all chunks, a private (random) value x_i is required. To create the signature of one chunk σ_i, the hash function is applied m_i times on the corresponding private value $\sigma_i = h^{m_i}(x_i)$. An example with $w = 4$ is illustrated in Figure 1.

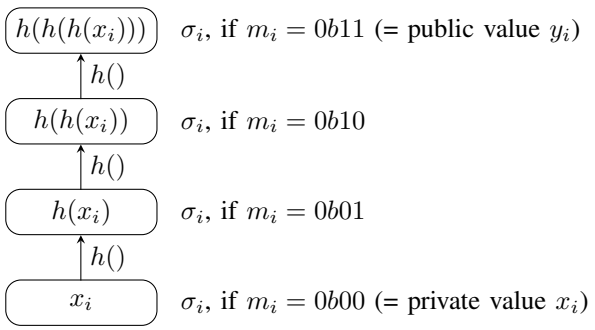

Fig. 1. WOTS construction with Winternitz parameter $w = 4$ signs 2 bits per value σ_i.

Because σ_i is public, anyone could create a signature for any sub-message between m_i and $w-1$. This issue is addressed by an additional checksum, which is signed in the same manner as the message itself. The checksum is the number of sub-messages times $w - 1$ minus the sum of all sub-message chunks. This prevents anyone to create a second valid signature if any message bit is changed.

To derive the public key from the private key, the hash function is applied $w - 1$ times on all private values $y_i = h^{w-1}(x_i)$. By applying the hash function $w - m_i - 1$ times on a signature value, the public key can be derived from a signature and the corresponding message: $y_i = h^{w-m_i-1}(\sigma_i)$.

Later improvements replaced the pure hash function by a chain function [18]. A pseudo-random value (mask) with length n is logical-XOR linked with the hash input to suppress collision-resistance requirements of the underlying hash function [19]. This one-time signature construction is referenced to as WOTS$^+$.

The downside of both WOTS and WOTS$^+$ is that if two different messages are signed using the same key pair, its security is broken [20].

B. Merkle Tree

A useful structure to arrange keys efficiently is the Merkle tree. 2^d WOTS$^+$ key pairs are merged in a balanced binary tree of depth d. The tree leaves $N_{i,0}$ are derived from the WOTS$^+$ public key by applying the hash function $N_{i,0} = h(Y_i)$, where Y_i is the concatenation of all WOTS$^+$ public values. The tree nodes $N_{i,j}$ are digests of its concatenated child nodes (a pair of neighbor leaves or inner nodes) $N_{i,j} = h(N_{2i,j-1}||N_{2i+1,j-1})$. The public key of all 2^d signatures is represented by the root node $N_{0,d}$. A signature itself contains a leaf index $0 \leq i < 2^d$, the WOTS$^+$ signature σ_i, and the authentication path. The latter includes all required tree nodes which verifiers need to calculate the public key $N_{0,d}$ by themselves.

To verify the signature, Y_i and its signature σ_i are derived from the message. Then, leaf $N_{i,0}$ is generated using Y_i. Using $N_{i,0}$ and the authentication path, the verifier is then able to calculate the root. If the result matches the signer's public key ($N_{0,2}$), the signature is valid.

A Merkle-tree improvement that significantly decreased the private key size was proposed by Buchmann et al. in [21]. Instead of storing all private values $x_{i,j}$, only a single seed is used. All private values are derived from this seed using a pseudo-random function. Fractal trees [22] are a further improvement. If a Merkle tree is large, key generation is very expensive in processing effort. All leaves and nodes must be calculated to get the public key (i.e. the tree root). A fractal tree (also referred to as hypertree) cuts this huge Merkle tree into several sub-trees. The lowermost Merkle tree is used to sign the message, and all upper sub-trees are used to sign the lower sub-tree roots.

All mentioned improvements lead to the extended Merkle signature scheme (XMSS) [12], which is a state-based construction. Because the same WOTS$^+$ private key must not be used twice, the signer must keep track of the state under all circumstances. The state consists of a scratch list of all used WOTS$^+$ keys. Depending on the system, this requirement may be hard to reach: Imagine, for instance, system crashes, resets, or parallelization setups (i.e., different devices using the same private key).

C. Forest of Random Subsets FORS

In principle, FORS [6] is a variant of Lamport's one-time signature (OTS) [16], but with a large security margin. This margin ensures that the same FORS key pair can be used to sign a few different messages. This security margin is paid for with a larger signature size and increased processing time.

FORS uses two parameters, k and $t = 2^a$, to sign bit strings of length $k \cdot a$. FORS needs $k \cdot t$ private values of length n, which are grouped into k sets. Each set contains t private values. To get the public values, a hash function is applied to all private values. Each set of public values is filled in a binary hash tree of depth a. This way, the public values are compressed into k root values. These root values are further compressed into a single FORS public key of size n by applying the hash function to all concatenated root nodes.

Signing starts with calculating the message digest, which is cut into k bit strings of size a. These bit strings are interpreted as unsigned integer numbers. Each number is an index corresponding to a private value in one of the k sets. These private values and the corresponding authentication paths are put into the signature. In total, the FORS signature consists of k private values and $k \cdot a$ authentication path nodes. All these values have size n, which corresponds to the security level (under the condition that $a \cdot k > n$).

Public key generation comes almost for free during signing, because all k FORS trees are evaluated anyway. A verifier calculates the $k \cdot a$ private value indices by applying the hash function on the message. Then, the k FORS root nodes are derived from the k private values and its authentication paths. As during key generation, the verifier calculates the FORS public key by applying the hash function to all concatenated root nodes.

D. SPHINCS$^+$

SPHINCS$^+$ finally is a combination of FORS and a large fractal Merkle tree of size h, split in h/d subtrees. Figure 2 shows the whole structure. A SPHINCS$^+$ private key is essentially SK.`seed`, a seed of length n (generated as true random as possible) which is used to derive all private values. A SPHINCS$^+$ public key is basically PK.`root`, the root node of the highest WOTS$^+_{subtree}$.

Signing starts by selecting a (pseudo-) random start address (2^h choices). This address selects the FORS key pair, which is used to sign the message digest. The public key of the selected FORS key pair is signed with the fractal Merkle tree. The size of 2^h (in the order of 2^{64}) ensures that the probability to choose the same start address many times is very low. In combination with the fact that using the same FORS key (i.e. choosing the same start address) a few times only does not harm, a signer does not have to keep track of previously selected start addresses. This fact qualifies for "statelessness".

The price for statelessness is increased processing effort and larger signature size. A full FORS signature and h/d WOTS$^+$-based Merkle trees must be evaluated to sign one message. Signing requires (depending on parameters) up to ten million hash function calls. The time necessary for one hash function evaluation is therefore a major criterion of the SPHINCS$^+$ processing speed. The use of the following three different hash functions is proposed in the SPHINCS$^+$ paper [6]: SHA-256 (SHA-2), SHAKE256 (SHA-3) and Haraka. Both SHA-2 and SHA-3 have the advantage of being NIST approved and enjoy high user confidence. Haraka is chosen because it can be sped up in case of having an AES-accelerator available. Beside the different parameter sets and hash functions, a *robust* version with masks and a *simple* version without masks exist. In the following, we focus on the 12 SHAKE256-based SPHINCS$^+$ versions defined by six parameter sets (with differing computational effort, signature size, and security margin), each of which in a simple and robust version.

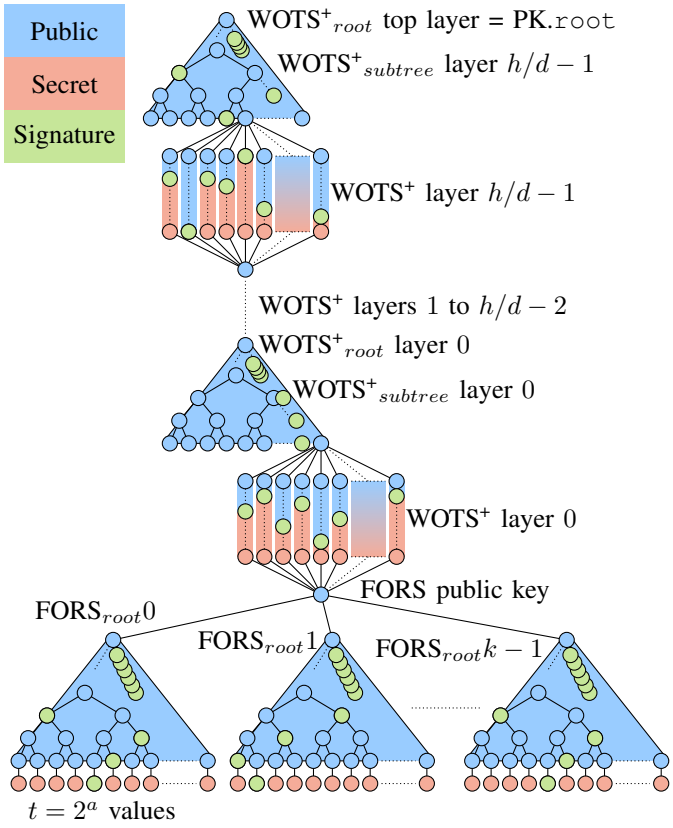

Fig. 2. A SPHINCS$^+$ signature consists of a FORS signature (which signs the message) and several WOTS$^+$ signatures and authentication paths (which sign the FORS public key).

III. FPGA Implementation

This section describes our FPGA-based hardware architecture for the SPHINCS$^+$ scheme. It follows similar design philosophies as in our previous SPHINCS-256 implementation, which we described in [9]. In the first subsection, similarities are summarized and the differences are clarified. The second subsection describes the top-level architecture and important sub-blocks. The section is completed with an analysis of the SPHINCS$^+$ performance, which is compared to software-based SPHINCS$^+$ and FPGA implementations of other post-quantum signature schemes.

A. From SPHINCS-256 to SPHINCS$^+$

The SPHINCS-256 FPGA architecture [9] consists of five main blocks: A control unit generates one internal instruction per clock. A fully unrolled hash core (fast, but large) computes one ChaCha12 hash result per clock cycle (ChaCha12 and π_{ChaCha} are primary hash functions in SPHINCS-256). A second iterative hash core (small, but slower) computes BLAKE-256 (only a small number of BLAKE-256 evaluations are required to generate a signature in SPHINCS-256). A memory block accessible from the trusted host contains the signature and keys (including static masks). A second memory block is used to buffer intermediate results such as tree nodes.

The main concept of computing one internal instruction per clock cycle has been kept. Also, the structure with a control unit generating internal instructions, two memory blocks, and a fully unrolled pipeline computing the hash function has remained. However, the algorithmic changes from SPHINCS-256 to SPHINCS$^+$ required some adjustments in the hardware architecture.

All parts in the SPHINCS$^+$ algorithm use the same hash function, in our case SHAKE256. Therefore, the ChaCha12 block has been replaced by the SHAKE256 block, and the second hash block in the SPHINCS-256 core (BLAKE-256) has been eliminated. A second notable difference is the way how masks are used. SPHINCS-256 uses level-based masks stored in the keys, whereas SPHINCS$^+$-robust derives the masks from the public seed and an address. This approach significantly reduces key sizes. A mask is calculated by an additional SHAKE256 evaluation. Because the individual mask is a part of the SHAKE256 block input, the mask must be calculated beforehand. To avoid pipeline stalls, masks are calculated in batches and buffered in memory. The buffer is realized in a first-in-first-out (FIFO) fashion. Since SPHINCS$^+$-simple does not use masks at all, the mask FIFO is not used for these instances. A third notable difference concerns the way how the start address is picked. Whereas the SPHINCS-256 FPGA architecture leaves this part as well as the hashing of the message to the host, our new SPHINCS$^+$ implementation takes care of both.

B. SPHINCS$^+$ Architectural and Design Decisions

The resulting SPHINCS$^+$ co-processor is pictured in Figure 3. The architecture includes:

Control unit	A state machine in charge of generating internal instructions
SHAKE256	A pipelined SHAKE256 hash value calculator
I/O RAM	Key and signature memory
Cache RAM	Memory for intermediate results
Mask FIFO	Buffer for masks (SPHINCS$^+$-robust only)

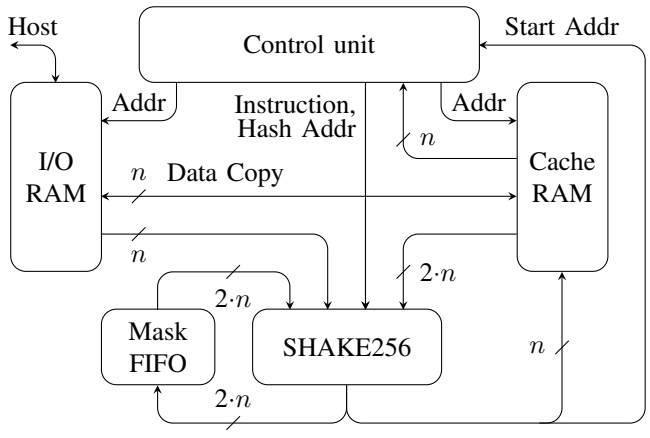

Fig. 3. SPHINCS$^+$ co-processor architecture. Depending on the instance, n is either 128, 196, or 256.

Both RAM blocks are instantiated as simple dual-port RAMs using the block RAM (BRAM) available in the FPGA device. The I/O RAM contains an additional switch (mux) to enable both internal and external read and write. The I/O RAM contains 2048 words and the cache RAM 1024 words of size n (= 128, 192, or 256).

The control unit consists of several interdependent finite state machines (FSM). All SPHINCS$^+$ algorithms are modeled in VHDL. This includes the computation of address indices (which are part of the SHAKE256 input) and read and write addresses for RAMs. The main control unit is designed to create one internal instruction per clock cycle. Most internal instructions are then processed in three steps: Read input data from RAM, execute SHAKE256, and store the output data into RAM.

Because the main difference between the SPHINCS$^+$ and SPHINCS-256 architecture lies in the choice of the hash function, the SHAKE256 core is described in more detail in the following sections.

C. SHAKE256 Core

SHAKE256 is part of the SHA-3 NIST standard [23]. As an extendable output function (XOF), it acts like a hash function with arbitrary output length. The core of all SHA-3 functions is the 1600-bit wide KECCAK permutation, which is called 24 times during the execution of SHAKE256.

Several FPGA-based SHA-3 implementations with different design goals are reported in the literature. The main difference is the target throughput resulting in a specific amount of required resources. Reported throughput-area trade-offs are categorized in [24]. The groups comprise:

Basic	This category covers straightforward implementations where one round per clock cycle is computed.
Folded	The round computation is divided into several clock cycles while processing a piece of the state every clock cycle. This technique enables the most compact implementations.
Pipelined	The parallel round computation is split into multiple clock cycles to enable higher clock frequencies. This allows to process multiple messages simultaneously.
Unrolled	Several round pipelines are instantiated and connected serially. This technique allows for the highest throughput.

The SHAKE256 throughput is crucial for the overall performance of the SPHINCS$^+$ core. Therefore, an unrolled structure fits best. The KECCAK round computation itself is implemented in the following pipeline structure:

$\theta_{1/2}$	Five 64-bit intermediate values are calculated by applying $\oplus$ (XOR) to five words of the KECCAK state each. This requires 1,600 + 320 flip-flops (FFs) and 320 5-input lookup tables (LUTs).
$\theta_{2/2}$	The second part of θ includes a one-bit rotation (comes for free in the pipeline as it is fully integrated into the routing) and the application of $\oplus$ twice on all 1,600 state bits using the five intermediate values. This part requires 1,600 3-input LUTs and 1,600 FFs.

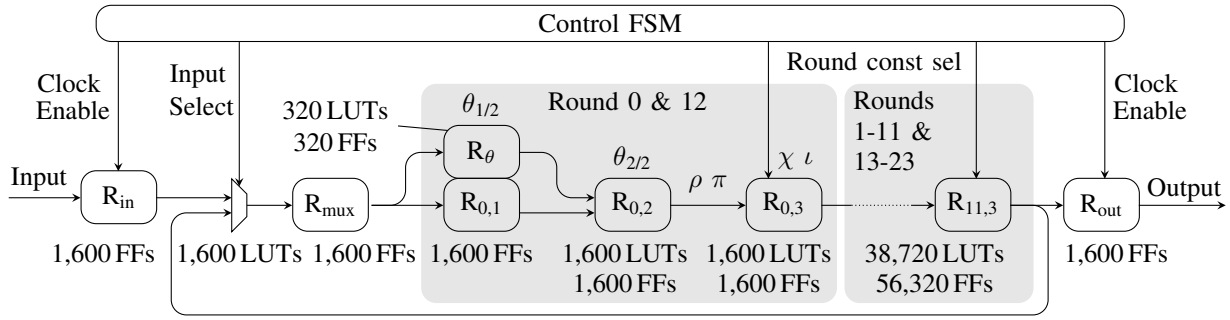

Fig. 4. The KECCAK pipeline runs at double clock speed and is executed twice per hash computation. That way, the pipeline depth is reduced from 24 to 12 rounds of KECCAK, saving almost 50 % of FPGA resources.

- $\rho\pi$ The cyclic rotating (ρ) and internal state reordering (π) comes for free as they are absorbed into the routing.
- $\chi\iota$ To perform χ, all 1,600 state bits are computed by a three-input Boolean function. The final ι step modifies one state word by applying $\oplus$ with a round constant (e.g. a 64-bit value depending on the round index i_r). The third pipeline stage consumes 1,600 3-input LUTs and 1,600 FFs.

To fully unroll SHAKE256, the round pipeline must be instantiated 24 times. This design choice would require a massive amount of FPGA resources. As a trade-off between area and throughput, the unrolling factor is cut in half, from 24 down to 12. That way, each KECCAK permutation runs through the 12-round long pipeline twice. The pipeline re-entry requires an additional 2-input mux to switch the pipeline input every clock cycle between a new input and the current pipeline output. The structure is visualized in Figure 4. The resulting KECCAK permutation pipeline is able to process a message on every other clock cycle. A single permutation has a delay of 72 clock cycles while 36 different permutations can be evaluated in parallel.

Our SPHINCS$^+$ architecture is designed to handle one hash input and output on every clock cycle. Therefore, the clock within the KECCAK pipeline has been chosen to run at double speed compared to the rest of the core. Because the KECCAK clock frequency is an integer multiple (factor 2) of the main clock frequency, clock domain crossing can be achieved synchronously.

The SHAKE256 core is built around the KECCAK pipeline. It contains some logic for input formatting and needs to handle cases when the KECCAK permutation is called multiple times. This happens whenever either the input or the output is larger than the SHAKE rate of 1088 bits (136 bytes). During SPHINCS$^+$ signing and verification, this happens for instance when the 67 WOTS$^+$ public key values are compressed into a single tree leaf.

IV. RESULT ANALYSIS

All proposed SHAKE256-based SPHINCS$^+$ instances have been implemented in 7-series Xilinx FPGAs. All presented results were obtained after place-and-route using the Xilinx Vivado 2018.2 tool with fully closed timing. The correct functionality was verified by execution on a KC705 evaluation board. This board is populated with the Kintex-7 XC7K325T-2 FPGA device. On this device, the main clock frequency is 300 MHz, and the KECCAK permutation clock runs at 600 MHz. Because NIST recommends Artix-7 to report the performance of algorithms in the post-quantum standardization process, detailed results are also reported for the Artix-7 device XC7A100T-3. On this device, the main clock runs at 250 MHz and the KECCAK permutation at 500 MHz.

The main part of the SPHINCS$^+$ FPGA implementation consists of the SHAKE256 core. Therefore, its performance is reported and compared to other implementations separately in the first part of this section. The performance results of the full core are listed in the second part of this section.

A. SHA-3 Core Performance

The SHAKE256 core including input and output formatting, flow control, and clock domain crossing roughly includes 45,500 LUTs and 70,000 FFs. Our SHAKE256 core runs at a 250 MHz clock (internal KECCAK permutation runs at 500 MHz) on the Artix-7 device and is able to compute one result per clock cycle. In the single-block message case, where both the message and output data sizes are less than 1,088 bits, the throughput reaches 250 MHz · 1,088 bits = 272 Gbps. The latency of a single message adds up to 41 clock cycles or 164 ns. The often-used throughput/area metric results in roughly 19 Mbps/slice. The performance is summarized in Table I.

TABLE I
PERFORMANCE RESULTS FOR SHA-3 KECCAK

Ref, device	Area slices	Frequency MHz	TP Gbps	TP/Area Mbps/slice
[25] Virtex-6	91	311	0.2	2.23
[26] Virtex-7	1,618	434	20.8	12.9
This, Artix-7	14,354	250 & 500	272	18.9
This, Kintex-7	15,049	300 & 600	326	21.7

The highest throughput/area metric we found in the open literature is 12.9 Mbps/slice for a Virtex-7 device [26]. Even though we use a slower FPGA, our highly pipelined SHAKE256 implementation is almost 70 % more efficient

TABLE II
SPHINCS$^+$-SHAKE256 PERFORMANCE (FOR ARTIX-7 XC7A100T-3 FPGA) COMPARED TO OTHER FPGA IMPLEMENTATIONS

Scheme, reference	Device	NIST Sec. Level	Signature kbyte	Area LUT	Area FF	Area BRAM	Area DSP	f_{clock} MHz	t_{sign} ms	t_{verify} ms	Power W	E_{sign} mWs	E_{verify} mWs
SPHINCS$^+$-128s-simple	Artix-7	1	8.1	48,231	72,514	11.5	0	250 & 500	12.4	0.07	9.71	120	0.7
SPHINCS$^+$-128s-robust	Artix-7	1	8.1	49,146	73,069	15.5	0	250 & 500	21.1	0.11	9.87	208	1.1
SPHINCS$^+$-128f-simple	Artix-7	1	17	47,991	72,505	11.5	1	250 & 500	1.01	0.16	9.76	9.9	1.5
SPHINCS$^+$-128f-robust	Artix-7	1	17	48,930	73,002	15.5	1	250 & 500	1.64	0.23	9.94	16.3	2.3
SPHINCS$^+$-192s-simple	Artix-7	3	17.1	48,725	72,514	17	0	250 & 500	21.4	0.10	9.81	210	1.0
SPHINCS$^+$-192s-robust	Artix-7	3	17.1	50,064	74,462	22.5	0	250 & 500	38.3	0.15	9.93	380	1.5
SPHINCS$^+$-192f-simple	Artix-7	3	35.7	48,398	73,476	17	1	250 & 500	1.17	0.19	9.69	11.4	1.8
SPHINCS$^+$-192f-robust	Artix-7	3	35.7	47,227	74,279	22.5	1	250 & 500	2.12	0.31	10.2	21.7	3.1
SPHINCS$^+$-256s-simple	Artix-7	5	29.8	51,130	74,576	22.5	1	250 & 500	19.3	0.14	9.75	188	1.3
SPHINCS$^+$-256s-robust	Artix-7	5	29.8	50,070	75,738	30	1	250 & 500	36.1	0.20	10.3	373	2.1
SPHINCS$^+$-256f-simple	Artix-7	5	49.2	51,009	74,539	22.5	1	250 & 500	2.52	0.21	9.80	24.7	2.0
SPHINCS$^+$-256f-robust	Artix-7	5	49.2	50,341	75,664	30	1	250 & 500	4.68	0.34	10.2	47.7	3.4
SPHINCS-256 [9]	Kintex-7	*	41	19,067	38,132	36	3	525	1.53	0.07	4.97	7.6	0.5
XMSS-2^{16} SHA-256 [11]	Cyclone V	*	2.7	6,500	9,540	145	0	93.1	9.95	5.80	Not provided		
Picnic-L1-FS [27]	Kintex-7	1	34	90,037	23,105	52.5	0	125	0.25	0.24	Not provided		
Picnic-L5-FS [27]	Kintex-7	5	133	167,530	33,164	98.5	0	125	1.24	1.17	Not provided		
qTESLA-p-I [28]	Artix-7	1	2.6	Inconclusive, ca. 2440 slices				121	34.4	7.8	Not provided		
qTESLA-p-III [28]	Artix-7	3	5.7	Inconclusive, ca. 2470 slices				121	63.9	19.1	Not provided		

*Security level not defined, because signature scheme is not in NIST post-quantum standardization process.

than the SHA3-224 implementation in [26]. This improvement stems from the higher clock frequency (three pipeline stages per round instead of two) and the comparatively reduced amount of multiplexers (gained by unrolling).

Compact, folded FPGA implementations of the KECCAK permutation have often throughputs up to a few hundred Mbps. Such an implementation is reported in [25] and requires 359 LUTs and 107 FFs. The architecture attains a throughput of 203 Mbps on a Virtex-6 FPGA and the throughput/area metric is reported to be 2.23 Mbps/slice.

B. SPHINCS$^+$ Core Performance

All measured times for signing and signature verification and the required FPGA resources are summarized in Table II. Additionally, the results are visualized in Figure 5. In all SPHINCS$^+$ instances, the critical elements are LUTs (i.e. highest utilization of available resources within the FPGA).

Although the key generation algorithm is not directly implemented, the SPHINCS$^+$ co-processor can be used to accelerate the key generation. It requires random data to generate the seeds (SK.`prf`, SK.`seed`, and PK.`seed`), followed by the computation of the top-most WOTS$^+$ subtree to get the public root (PK.`root`). This root calculation comes for free during signing. If random data is available, the signing function can be called to get the public root (signature data is ignored).

Compared to the SPHINCS-256 implementation [9], the SPHINCS$^+$ core needs twice the amount of FFs and 2.5 times more LUTs. This originates primarily from the hash function. Without the SHAKE256 block, our implementation occupies (depending on the instantiation) 3,000 to 6,000 LUTs. The SPHINCS-256 core from [9] needs $\approx$5,000 LUTs without hash blocks (ChaCha12 and BLAKE-256). Besides the area requirement, our SPHINCS$^+$ implementation is also slower: The clock speed is 2.1 times lower in the SPHINCS$^+$ co-processor. The SPHINCS$^+$ co-processor's main clock runs with 250 MHz, while the SPHINCS-256 core runs at 525 MHz.

Compared to the software-based SPHINCS$^+$-SHAKE256 benchmark reported in [6], our SPHINCS$^+$ core achieves a speed-up factor in signing of around 100 compared to the reference implementation running on a 3.5 GHz Intel i7 processor. Compared to the optimized implementation using AVX2 instructions, our core is around 50 times faster in signing. If SPHINCS$^+$-Haraka with a hardware accelerator for Haraka is taken into account, our SPHINCS$^+$ core is still around 8 times faster. The corresponding factors for verifying are 50, 25, and 3, respectively.

Additional FPGA implementations have been reported for state-based signature schemes. An interesting design is described in [11], where an XMSS [12] hardware accelerator is integrated in a RISC-V processor. The SHA-256 XMSS version with $h = 16$ (enables 2^{16} signatures) occupies 6,500 ALMs (8-input LUTs), 9,540 FFs, 145 BRAMs and attains 9.95 ms (average for the first 16 signatures), 5.8 ms to verify, and 3.44 s for key generation. If the high amount of BRAMs in [11] is omitted, our SPHINCS$^+$ core would require roughly seven times more resources. All SPHINCS$^+$ instances

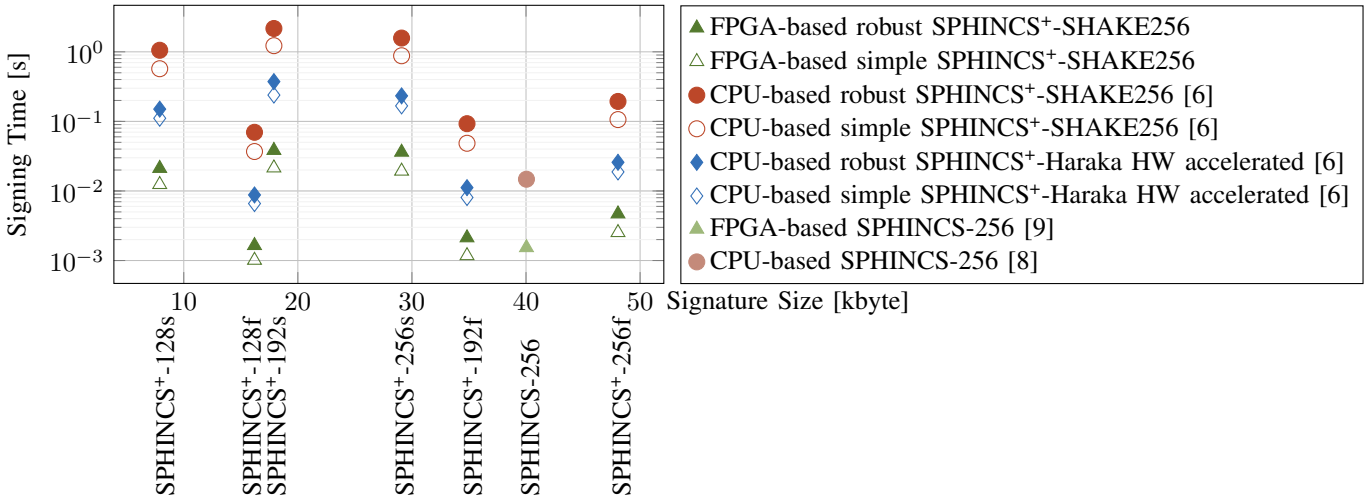

Fig. 5. The proposed FPGA-based SPHINCS⁺ co-processor implemented on Artix-7 compared to the processor-based implementation from [6].

are ten times faster in verification and more than 100 times faster for key generation. Signing delay of the SPHINCS⁺ core depends on the instance, while the XMSS signing delay even fluctuates depending on the processed key-pair number. However, with roughly 10 ms, the signing delays of both implementations are in the same ballpark.

FPGA implementations for Picnic, a signature scheme currently in NIST post-quantum standardization, are described in [27]. Our SPHINCS⁺-256f-simple implementation requires only one third of the FPGA resources than the Picnic level 5 implementation reported in [27]. Although the Picnic implementation attains half the signing time, the delay-times-area product is 50% higher than our SPHINCS⁺-256f-simple implementation. Concerning signature verification, our SPHINCS⁺-256f-simple implementation is 25 times more efficient in the delay-times-area metric.

Another signature scheme currently being reviewed by NIST is qTESLA. A hardware-software co-design implementation for qTESLA is reported in [28]. The architecture consists of a RISC-V processor with additional hardware-accelerated instructions. Although the required FPGA resources are provided for the individual hardware instructions, the required area of the full qTESLA architecture, including RISC-V processor and associated memory, is not reported. However, compared to the qTESLA-p-I implementation described in [28], our SPHINCS⁺-128s-simple implementation is three times faster in signing and 100 times faster in signature verification.

V. Fault Attack

Recent publications show that SPHINCS⁺ and other multi-tree-like hash-based signature schemes are susceptible to a fault attack. The attack was proposed by Castelnovi et al. in [29] and practically applied on an Arduino board by Genêt et al. [30]. So far, the attack has been applied on software implementations only. In this section, we present the first fault attack on a hardware implementation.

A. Attack Description

The fault attack is most effective if the start address (i.e. the FORS key pair selection) can somehow be controlled. The prerequisite is that the authentication path is partly shared for several signatures. This requirement is not mandatory for success, it just accelerates the attack. An attacker without influence on the signed message can utilize the attack anyway. In this case, the attacker just has to wait until enough signatures are created with the same authentication path in the topmost WOTS⁺$_{subtree}$ (this happens in average at every 64$^{\text{th}}$ signature for SPHINCS⁺-256s).

For simplicity, let us assume that the same message is signed multiple times and the optional random input is switched off. The SPHINCS⁺ algorithm will produce exactly the same signature on every call. The topmost WOTS⁺ signature at layer $h/d-1$ signs the WOTS⁺$_{subtree}$ root node of layer $h/d-2$. This WOTS⁺ signature releases some private nodes of its private key. The released information cannot be utilized to sign another message. If an attacker manages to change the value of the WOTS⁺$_{subtree}$ root node at layer $h/d-2$, the WOTS⁺ private key at layer $h/d-1$ is used to sign this tampered root node. The resulting signature releases then some additional private key nodes. The attack is visualized in Figure 6.

One correct and one tampered SPHINCS⁺ signature with a partly identical authentication path leave a complexity of 2^{34} hash function computations to forge a signature [20]. However, the attacker can repeat the attack until he recovers the full private key of WOTS⁺ at layer $h/d-1$. A detailed analysis on the security margin depending on the number of faulty signatures is presented in [20].

To apply the attack, an attacker must be able to provoke a random fault during WOTS⁺ public key calculations. To accurately time the fault attack, a passive side-channel attack can be applied. An attacker could also use a try-and-error approach because the timing is not tight. As long as any of the WOTS⁺ public keys at layer $h/d-2$ has a flipped bit, the

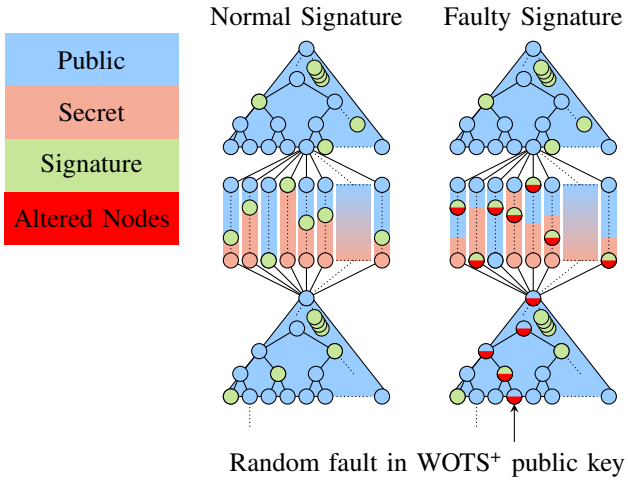

Fig. 6. A fault during WOTS$^+_{subtree}$ calculations leads to a different root node value. This wrong root node is signed with the next level of WOTS$^+$ and leaking therefore some private values of the corresponding WOTS$^+$ private key.

attack will work. The example in Figure 6 shows the effect of a faulty WOTS$^+$ public key. Not only the root node but also one node in the authentication path (part of the SPHINCS$^+$ signature) has an altered value. Because the change in the root node is a function of the changed node in the authentication path, the exact same change in the root node results when the verification algorithm is called. This leads to the behavior that the SPHINCS$^+$ signature remains valid even if a fault has been injected.

These circumstances make the fault attack very powerful. A random single bit flip is sufficient to break the scheme. If the fault attack is correctly timed, the faulty SPHINCS$^+$ signature is still valid. A sign-then-verify function does therefore not detect a successful fault attack.

B. Attacking the FPGA Implementation

To apply the attack, we must be able to inject faults into the FPGA. One approach is to add variations to the power supply. A reduced supply voltage causes internal digital circuits to have larger gate delays [31], which may lead to timing violations. This violation can provoke a computation error. Injecting a fault into an FPGA by temporarily decreasing the voltage has already been exploited in [32] and [33].

By applying supply-voltage glitches, an attack against the SPHINCS$^+$ scheme described above was successfully performed on our FPGA implementation.

The number of faulty signatures required to get the entire WOTS$^+$ private key is shown in Figure 7. The theoretical expected value of the number of needed signatures ($\mathbb{E}(X) \approx 74.5$ [29]) is consistent with our measurement ($\mathbb{E}(X) \approx 74$). With our attack setup, it is possible to obtain the entire WOTS$^+$ private key in about 70 seconds.

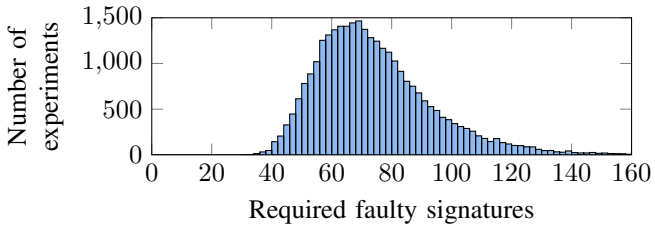

Fig. 7. Required number of faulty signatures to retrieve the whole WOTS$^+$ private key.

C. Countermeasures

Detecting faults in Merkle trees is an ongoing research challenge. Genêt et al. [30] propose a caching mechanism. Their approach is able to protect stateful schemes, but it is not fully applicable to stateless schemes such as SPHINCS$^+$. Kermani et al. [34] propose to detect fault attacks by calculating parts of the hypertree a second time with swapped processing order. This could be applied to our SPHINCS$^+$ core, but the signing time would increase due to the sequentially executed, redundant calculations.

Hardware implementations allow us to use protection techniques that are not available to software implementations. Instead of protecting the SPHINCS$^+$ core, a dedicated fault-attack detection logic could be implemented. An FPGA-based fault-attack detection sensor is presented in [35]. Such fault-detection circuits are reported to be efficient with respect to resource overhead and are reliable in fault detection of all known fault attacks. This approach may be the best choice for a specific instantiation. However, we are looking for a protected SPHINCS$^+$ implementation that does not depend on extra detection circuits.

The simplest attempt to detect faults is to duplicate parts of the SPHINCS$^+$ core and compare the outputs of both duplicated blocks. If only the SHAKE core is instantiated twice, 20 % fault coverage is reached in our setup (concerning supply voltage glitches). Duplicating other blocks, such as RAMs or the control unit, does not provide notable fault coverage. Only if the full SPHINCS$^+$ co-processor is instantiated twice, full fault coverage is reached. Although both cores are functionally identical, they differ in their electrical characteristics. Both cores are placed and routed in different regions within the FPGA. A supply-voltage glitch attack will most probably affect both cores, but it is highly unlikely that exactly the same bit-flip will occur. By delaying one of the cores by a few clock cycles, it is even more difficult for an attacker to provoke the same processing error on both cores. This successful countermeasure comes at the price of doubling the required resources in the FPGA.

VI. Conclusions

We presented the first hardware implementation of the signature scheme SPHINCS$^+$-SHAKE256 as described in the documents of the NIST post-quantum competition. The implementation needs roughly 50,000 LUTs and 75,000 FFs on

a 7-series FPGA from Xilinx. Key generation and signing takes 1 ms for the (fastest) variant SPHINCS$^+$-128f-simple, and 38.3 ms for the (slowest) variant SPHINCS$^+$-192s-robust. Verification takes between 0.07 ms and 0.34 ms. Experiments with supply voltage glitches have revealed that an FPGA implementation of the SPHINCS$^+$ scheme is prone to a fault attack. With a glitch attack, collecting private information to forge a signature is a matter of seconds. A successful countermeasure consists in doubling the entire SPHINCS$^+$ co-processor.

VII. Acknowledgments

This work was supported by Innosuisse, the federal agency responsible for encouraging science-based innovation in Switzerland.

References

[1] R. L. Rivest, A. Shamir, and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] National Institute of Standards and Technology, "Digital Signature Standard (DSS)," FIPS-PUB 186-4, 2009.

[3] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[4] National Institute of Standards and Technology, "Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process," 2016.

[5] G. Alagic *et al.*, "Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process," NISTIR 8240, 2019.

[6] A. Hülsing *et al.*, "SPHINCS+ Submission to the NIST post-quantum project," 2019.

[7] J. Buchmann, E. Dahmen, and M. Szydlo, "Hash-based Digital Signature Schemes," in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 35–93.

[8] D. J. Bernstein *et al.*, "SPHINCS: Practical Stateless Hash-Based Signatures," in *Advances in Cryptology - EUROCRYPT 2015*, ser. LNCS, vol. 9056. Springer, 2015, pp. 368–397.

[9] D. Amiet, A. Curiger, and P. Zbinden, "FPGA-based Accelerator for Post-Quantum Signature Scheme SPHINCS-256," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 18–39, 2018.

[10] S. Sun, R. Zhang, and H. Ma, "Efficient Parallelism of Post-Quantum Signature Scheme SPHINCS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2542–2555, 2020.

[11] W. Wang, B. Jungk, J. Wälde, S. Deng, N. Gupta, J. Szefer, and R. Niederhagen, "XMSS and Embedded Systems - XMSS Hardware Accelerators for RISC-V," *IACR Cryptology ePrint Archive*, vol. 2018, p. 1225, 2018.

[12] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme," RFC 8391, 2018.

[13] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, 1997, pp. 37–51.

[14] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[15] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[16] L. Lamport, "Constructing Digital Signatures from a One Way Function," SRI International, CSL-98, 1979.

[17] R. C. Merkle, "A Certified Digital Signature," in *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, 1989, pp. 218–238.

[18] A. Hülsing, "W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes," in *Progress in Cryptology - AFRICACRYPT 2013*, ser. LNCS, vol. 7918. Springer, 2013, pp. 173–188.

[19] M. Bellare and P. Rogaway, "Collision-Resistant Hashing: Towards Making UOWHFs Practical," in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, 1997, pp. 470–484.

[20] L. G. Bruinderink and A. Hülsing, ""Oops, I Did It Again" - Security of One-Time Signatures Under Two-Message Attacks," in *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Revised Selected Papers*, 2017, pp. 299–322.

[21] J. A. Buchmann, L. C. C. García, E. Dahmen, M. Döring, and E. Klintsevich, "CMSS - An Improved Merkle Signature Scheme," in *Progress in Cryptology - INDOCRYPT 2006*, ser. LNCS, vol. 4329. Springer, 2006, pp. 349–363.

[22] M. Jakobsson, F. T. Leighton, S. Micali, and M. Szydlo, "Fractal Merkle Tree Representation and Traversal," in *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference, Proceedings*, 2003, pp. 314–326.

[23] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," FIPS PUB 202, 2015.

[24] M. Sundal and R. Chaves, "Efficient FPGA Implementation of the SHA-3 Hash Function," in *2017 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2017, Bochum, Germany, July 3-5, 2017*, 2017, pp. 86–91.

[25] B. Jungk and M. Stöttinger, "Hobbit - Smaller but Faster than a Dwarf: Revisiting Lightweight SHA-3 FPGA Implementations," in *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2016, Cancun, Mexico, November 30 - Dec. 2, 2016*, 2016, pp. 1–7.

[26] G. Athanasiou, G. Makkas, and G. Theodoridis, "High Throughput Pipelined FPGA Implementation of the new SHA-3 Cryptographic Hash Algorithm," in *6th International Symposium on Communications, Control and Signal Processing, ISCCSP 2014, Athens, Greece, May 21-23, 2014*, 2014, pp. 538–541.

[27] D. Kales, S. Ramacher, C. Rechberger, R. Walch, and M. Werner, "Efficient FPGA Implementations of LowMC and Picnic," in *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, ser. Lecture Notes in Computer Science, S. Jarecki, Ed., vol. 12006. Springer, 2020, pp. 417–441.

[28] W. Wang, S. Tian, B. Jungk, N. Bindel, P. Longa, and J. Szefer, "Parameterized Hardware Accelerators for Lattice-Based Cryptography and Their Application to the HW/SW Co-Design of qTESLA," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 54, 2020.

[29] L. Castelnovi, A. Martinelli, and T. Prest, "Grafting Trees: A Fault Attack Against the SPHINCS Framework," in *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, 2018, pp. 165–184.

[30] A. Genêt, M. J. Kannwischer, H. Pelletier, and A. McLauchlan, "Practical Fault Injection Attacks on SPHINCS," *IACR Cryptology ePrint Archive*, vol. 2018, p. 674, 2018.

[31] R. Ahmadi and F. N. Najm, "Timing Analysis in Presence of Power Supply and Ground Voltage Variations," in *2003 International Conference on Computer-Aided Design, ICCAD 2003, San Jose, CA, USA, November 9-13, 2003*, 2003, pp. 176–183.

[32] D. R. E. Gnad, F. Oboril, and M. B. Tahoori, "Voltage Drop-Based Fault Attacks on FPGAs Using Valid Bitstreams," in *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, 2017, pp. 1–7.

[33] J. Krautter, D. R. E. Gnad, and M. B. Tahoori, "FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 44–68, 2018.

[34] M. M. Kermani, R. Azarderakhsh, and A. Aghaie, "Fault Detection Architectures for Post-Quantum Cryptographic Stateless Hash-Based Secure Signatures Benchmarked on ASIC," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 2, pp. 59:1–59:19, 2017.

[35] W. He, J. Breier, and S. Bhasin, "Cheap and Cheerful: A Low-Cost Digital Sensor for Detecting Laser Fault Injection Attacks," in *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, 2016, pp. 27–46.