Streaming SPHINCS+ for Embedded Devices at the Example of TPMs

Ruben Niederhagen[1], Johannes Roth[2] and Julian Wälde[3]

[1] University of Southern Denmark
ruben@polycephaly.org
[2] MTG AG
johannes.roth@mtg.de
[3] Fraunhofer SIT
julianwaelde@gmail.com

Abstract. We present an implementation of the hash-based post-quantum signature scheme SPHINCS$^+$ that enables heavily memory-restricted devices to sign messages by streaming-out a signature during its computation and to verify messages by streaming-in a signature. We demonstrate our implementation in the context of Trusted Platform Modules (TPMs) by proposing a SPHINCS$^+$ integration and a streaming extension for the TPM specification. We evaluate the overhead of our signature-streaming approach for a stand-alone SPHINCS$^+$ implementation and for its integration in a proof-of-concept TPM with the proposed streaming extension running on an ARM Cortex-M4 platform. Our streaming interface greatly reduces the memory requirements without introducing a significant performance penalty. This is achieved not only by removing the need to store an entire signature but also by reducing the stack requirements of the key generation, sign, and verify operations. Therefore, our streaming interface enables small embedded devices that do not have sufficient memory to store an entire SPHINCS$^+$ signature or that previously were only able to use a parameter set that results in smaller signatures to sign and verify messages using all SPHINCS$^+$ variants. Since the streaming concept aggravates fault attacks on hash-based signature schemes, we briefly discuss countermeasures to attenuate such attacks in a signature-streaming scenario.

Keywords: SPHINCS+ · PQC · Signature Streaming · TPM · ARM Cortex-M4.

1 Introduction

Experts predict that within the next decade quantum computers may be available that will be capable of breaking current asymmetric cryptography based on the integer factorization and discrete logarithm problems by applying Shor's algorithm. This creates a significant pressure on academia and practitioners to propose, analyze, implement, and migrate to new cryptographic schemes that are able to withstand attacks aided by quantum computing. Such schemes are jointly referred to as Post-Quantum Cryptography (PQC). The research effort on PQC has culminated in a standardization process by the National Institute of Standards and Technology (NIST) in the US that started in December 2016 and is expected to be finished in 2024[1].

Currently, at the time of writing this paper in summer 2021, the NIST standardization process is in the third round with seven finalists that might get standardized as as result of this round and eight alternate candidates that might either replace finalists if

[1]https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline

vulnerabilities or implementation issues were to be found or that may be considered for future standardization in a possible fourth round. One of these alternate candidates is the hash-based PQC signature scheme SPHINCS$^+$ [HBD$^+$20].

Among the alternate candidates, SPHINCS$^+$ has a somewhat special role: NIST has stated in the 2nd round report [MAA$^+$20] that "NIST sees SPHINCS$^+$ as an extremely conservative choice for standardization" and that if "NIST's confidence in better performing signature algorithms is shaken by new analysis, SPHINCS$^+$ could provide an immediately available algorithm for standardization at the end of the third round." This statement has been emphasised also in a posting to the NIST mailing list by NIST on January 21, 2021[2].

Two major obstacles for the widespread adoption of SPHINCS$^+$ and also the reason why SPHINCS$^+$ was not selected by NIST as finalist in round 3 are its relatively large signature sizes and the relatively high computational signing cost. These costs are not prohibitive on larger computing systems like servers, PC, notebooks, tablets, and smart phones — but they might be a burden for small embedded devices with small computing power and small memory.

An example for such small devices are Trusted Platform Modules (TPMs). TPMs are Hardware Security Modules (HSMs) that are tightly coupled with a computing system and that provide an anchor of trust for sensitive applications. TPMs have a small protected and persistent memory for the secure and temper-protected storage of secret encryption and signature keys as well as public verification keys. Furthermore, TPMs can act as cryptographic engines that can encrypt and decrypt data as well as sign and verify messages using the securely stored key material. The idea is that secret keys never leave the TPM and that public keys can be stored securely and protected from manipulation to reliably authenticate communication partners. TPMs typically have a very limited amount of RAM and fairly limited computational capabilities.

Hash-based signature schemes come in two flavors: They can be *stateful*, meaning that a security-critical signature state needs to be maintained by the signer and updated after each signature computation, and they can be *state-free*, meaning that such a state does not need to be maintained. SPHINCS$^+$ is a state-free hash-based signature scheme while XMSS [BDH11] (published as an IETF-RFC [HBG$^+$18]) is a closely related example for a stateful scheme.

For TPMs, stateful signature schemes like XMSS might appear to be the better choice among hash-based signature schemes: On the one hand, signatures of state-free schemes like SPHINCS$^+$ are so large that they might not fit into the TPM memory and on the other hand, the tamper-proof and unclonable storage of TPMs mitigates many of the commonly stated concerns when dealing with stateful schemes. However, the persistent storage typically is implemented as flash memory that has a significant wear-down over time. If too many write operations are performed at the same memory address, errors might occur that would render stateful signature schemes insecure. Storing the sensitive state persistently and securely in a TPM hence is not a simple task and might pose a security risk. Therefore, using a state-less signature scheme like SPHINCS$^+$ might be more secure — if the larger signatures can be handled.

Depending on variant and parameter set, the size of SPHINCS$^+$ signatures ranges between 8 kB and 50 kB although private and public keys are very small with 64 B to 128 B and 32 B to 64 B respectively. Some embedded devices (including TPMs) do not have sufficient memory to store an entire signature during signing or verifying a message. As a solution to this problem, splitting a signature into several smaller parts and streaming these parts out of or into an embedded device has been proposed and evaluated for SPHINCS-256 signing and verification in [HRS16] and for SPHINCS$^+$ verification in [GHK$^+$21]. In this paper, we are refining this approach and we are evaluating it exemplarily on TPMs.

[2] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/2LEoSpskELs/m/VB1jng0aCAAJ

Contributions. Our contributions in this paper are:

- the integration of a signature streaming interface for signing and verification into the reference implementation of SPHINCS$^+$,
- a performance and stack-usage evaluation of this interface on ARM Cortex-M4,
- the design and prototypical implementation of a SPHINCS$^+$ and streaming extension for the TPM specification,
- the evaluation of this TPM-streaming protocol with our streaming modification of SPHINCS$^+$ on the Cortex-M4 platform, and
- a brief discussion of the impact of fault attacks on SPHINCS$^+$ in the context of this signature-streaming scenario with the proposal of a mitigating countermeasure.

The source code of our implementation together with the TPM prototype integration is available at `https://github.com/QuantumRISC/mbedSPHINCSplusArtifact` under BSD license.

Related Work. There is some work on computing hash-based signatures on embedded devices. Wang et al. propose hardware accelerators for XMSS on a RISC-V platform in [WJW$^+$19] and Amiet et al. introduce hardware accelerators for SPHINCS-256 in [ACZ18] (SPHINCS-256 is a predecessor of SPHINCS$^+$). Kölbl describes an implementation of SPHINCS on a relatively large ARMv8-A platform in [Köl18] and Campos et al. compare LMS and XMSS on an ARM Cortex-M4 in [CKR$^+$20]. The pqm4 project [KRS$^+$] provides performance measurements of several PQC schemes including SPHINCS$^+$ on a Cortex-M4 platform. All of these implementations assume that the embedded platform has sufficient memory to store an entire hash-based signature.

As mentioned above, the idea to stream signatures has been applied to SPHINCS$^+$ (or its predecessors) before. In [HRS16], an implementation is presented that computes a 41 kB signature of SPHINCS-256 [BHH$^+$15] on a Cortex-M3 processor with 16 kB of RAM. The authors show that key generation, signature generation, and signature verification are possible on the device by splitting the SPHINCS-256 signature into several relatively large parts, which are streamed separately. In [GHK$^+$21] Gonzalez et al. investigate signature and public key streaming for verification for several PQC schemes including SPHINCS$^+$ in only 8 kB of RAM using relatively large blocks for streaming. We extend and refine their work to cover also signing (and key generation) and we provide a more detailed analysis for more SPHINCS$^+$ parameter sets and for the use of signature streaming for TPMs.

Streaming of PQC schemes has also been discussed in other publications. For example, Gonzalez et al. also apply their streaming approach for signature verification to the PQC signature schemes Rainbow, GeMSS, Dilithium, and Falcon [GHK$^+$21]. Roth et al. in [RKK20] are streaming the large public key of the code-based scheme Classic McEliece into and out of a Cortex-M4 processor.

Structure of this paper. This paper is structured as follows: We provide an overview of SPHINCS$^+$ and TPMs in Section 2. We then describe the implementation of our streaming interface and its integration into a TPM prototype in Section 3 and evaluate our implementation in Section 4. We briefly discuss the impact of signature streaming for SPHINCS$^+$ on random fault attacks with potential countermeasures in Section 5 and conclude our paper.

2 Preliminaries

This section provides an introduction to the signature scheme SPHINCS$^+$ and to TPMs.

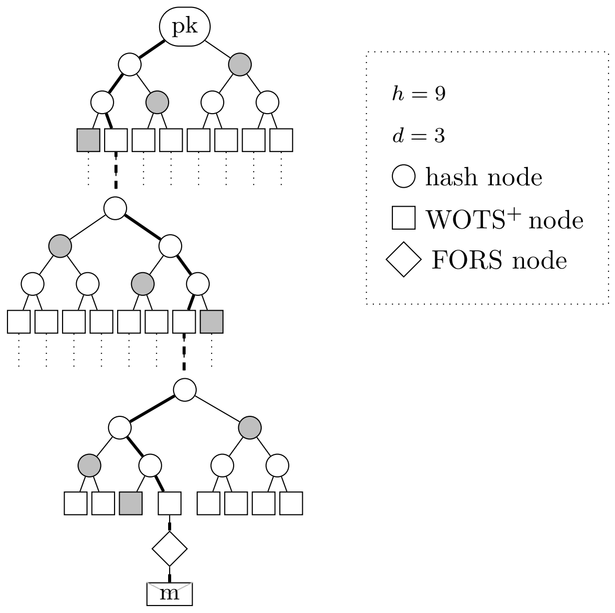

Figure 1: Illustration of a small SPHINCS$^+$ structure (figure from [BHK$^+$19]).

2.1 SPHINCS$^+$

The state-free hash-based post-quantum signature scheme SPHINCS$^+$ has been described in [BHK$^+$19] and has been submitted to the NIST PQC standardization process as defined in the submission document [HBD$^+$20]. It is composed of a hypertree of total height h with d layers of Merkle signature trees. These trees have WOTS$^+$ one-time signatures at their leaves such that the inner trees sign Merkle trees at the next hypertree layer and the leaves of the bottom layer sign FORS few-time signatures, which in turn are used to sign message digests. Figure 1 shows an example of the hypertree structure of SPHINCS$^+$ with reduced (hyper-)tree sizes for illustration. The output length n (in byte) of the underlying hash function defines the security level of a SPHINCS$^+$ signature.

Key generation of SPHINCS$^+$ is relatively expensive, since the entire first-layer tree of the hypertree including all WOTS$^+$ public keys on its leaf nodes needs to be computed in order to obtain the root node of the first-layer tree as public key.

The process of signing a message using SPHINCS$^+$ is very similar to signing with an XMSS multi-tree [BDH11; HBG$^+$18] with a notable distinction that XMSS does not use a few-time signature scheme at the bottom layer of the hypertree. For signing a message using SPHINCS$^+$, a FORS private key at the leaves of the trees at the bottom layer is selected and the message is signed with this few-time signature key. The resulting public key is then signed with the corresponding WOTS$^+$ one-time signature, an authentication path (the gray nodes in Figure 1) through the Merkle tree is computed as part of the signature, and the root node of the Merkle tree is obtained. This tree root is then signed using the corresponding WOTS$^+$ one-time signature of the corresponding tree at the next higher layer. The WOTS$^+$ public key is computed and again authenticated by computing an authentication path with the corresponding Merkle tree root. This process is repeated over the layers of the inner trees until finally the root tree of the hypertree is reached and the authentication path of the root tree have been computed.

Verification works equivalently: First, the FORS signature is used to compute a candidate for the FORS public key on the bottom layer of the hypertree. This candidate is then used to compute a candidate for the Merkle tree root of that bottom-layer tree using the WOTS$^+$ one-time signature and the authentication path. This root node is then used together with the WOTS$^+$ one-time signature and the authentication path on the next layer of the hypertree to compute a candidate for the Merkle-tree root at that layer and so forth. Finally, the candidate of the root node of the root tree of the hypertree is compared to the public key of the proclaimed signer of the message to verify the signature.

WOTS$^+$. The one-time signature scheme WOTS$^+$ is based on the idea to use hash chains (attributed by Merkle to Winternitz in [Mer90]). For key generation, random bit-strings of length n are generated and each hashed w times to obtain the end node of the chain. These end nodes are then concatenated and hashed to obtain the WOTS$^+$ public key. (Note that XMSS instead uses an unbalanced binary l-tree to compute the corresponding public key.)

For signing a message, the signer splits the message digest into words of w bits and computes a checksum over these words, which is split into w-length words as well. Each of these w-bit long words then serves as index into the hash chains: The signer iteratively recomputes the hash chain starting from the corresponding secret value until the node indexed by the w-bit word is reached and includes this value in the signature.

For verifying a WOTS$^+$ signature, the verifier recomputes the w-bit indices from the message digest and hashes each intermediate chain value in the signature up to the end node of that chain. The signature is valid if the hash over all concatenated end nodes is equal to the public key.

WOTS$^+$ is a one-time signature scheme since an attacker learns some of the chain nodes from a given signature. Due to the construction of the checksum over the w-bit words, a single signature does not enable an attacker to forge signatures. However, if the same public/private WOTS$^+$ key pair was used for several different signatures, an attacker would be able to learn a sufficient number of internal nodes and to eventually forge arbitrary messages.

FORS. FORS is short for Forest Of Random Subsets. It is a few-time signature scheme based on Merkle hash trees. FORS uses k trees of height a with $t = 2^a$ leaf nodes to sign a message digest of ka bits. The root nodes of the k trees are hashed together to obtain the FORS public key.

For FORS key generation, for each of the k trees t secrets of n-bytes are generated (temporarily from a secret seed using a derivation function). The public key is computed by hashing the secrets, by computing Merkle-tree nodes from the hashed secrets (using pairwise hashing), and by then concatenating and hashing the resulting k root nodes.

For generating a FORS signature, the signer computes a message digest of ka bits for the message, splits this digest into k a-bit strings, and interprets each of the a-bit strings as index into the corresponding tree. The secret indexed by each a-bit string is copied into the signature together with a verification path in the corresponding Merkle tree.

For verifying a FORS signature, the verifier recomputes the root nodes of the k trees by hashing the leaf node from the signature and using the verification path and obtains a public key candidate by hashing the root nodes together. This public key candidate is then authenticated by the SPHINCS$^+$ hypertree.

The parameters k and t are chosen according to the desired number of signatures that can be computed with this few-time signature scheme such that the secrets revealed in each signature do not provide enough information to an attacker in order to forge arbitrary signatures.

Hashing in SPHINCS$^+$. Hash-based signature schemes like SPHINCS$^+$ require a huge amount of hash-function calls: WOTS$^+$ is using many hash calls for computing the hash chains and the computations for FORS and Merkle trees are based on pairwise hashing of child nodes. Kannwischer et al. report in [KRS$^+$19] that 87 % to 98 % of the time in SPHINCS$^+$ operations is spent for hashing. Therefore, efficient hash-function implementations are important in order to achieve a high performance for SPHINCS$^+$.

Table 1: SPHINCS$^+$ parameter sets as proposed for NIST Round 3 [HBD$^+$20].

Parameter Set	n	h	d	$\log(t)$	k	w	Bit-security	Security Level	Signature [bytes]
SPHINCS$^+$-128f	16	66	22	6	33	16	128	1	17 088
SPHINCS$^+$-128s	16	63	7	12	14	16	133	1	7 856
SPHINCS$^+$-192f	24	66	22	8	33	16	194	3	35 664
SPHINCS$^+$-192s	24	63	7	14	17	16	193	3	16 224
SPHINCS$^+$-256f	32	68	17	9	35	16	255	5	49 856
SPHINCS$^+$-256s	32	64	8	14	22	16	255	5	29 792

Parameter sets. The parameters and parameter sets for SPHINCS$^+$ are shown in Table 1. The SPHINCS$^+$ specification offers a total of 36 parameter sets for three security levels, three different hash functions, "simple" and "robust" variants, and "small" as well as "fast" parameter sets. For the reminder of this paper we only consider variants that are using SHA-256 and SHAKE256 as hash functions.

The "s" (small) and "f" (fast) variants provide a trade-off between signature sizes and computational cost. The "s" parameter sets have a smaller number d of layers in the hypertree and hence taller subtrees of height h/d. Therefore, the "s" parameter sets provide smaller signatures (since fewer WOTS$^+$ signatures, one per layer, are required) at the cost of key generation and signing time (since the subtress are taller and exponentially more leaf nodes need to be generated). However, verification time is reduced since accordingly fewer WOTS$^+$ verifications are required. The "f" parameter sets significantly speed up key generation and signing while increasing signature sizes and verification time.

The security argument for the "simple" parameter sets involves an invocation of the Random Oracle Model (ROM) while the "robust" variants have a security proof that assumes pre-image resistance and the Pseudo Random Function (PRF) property for the hash-function constructions. The "robust" variants require more hash-function invocations, because additional inputs for tweakable hash functions need to be computed. This difference is equivalent to the difference between XMSS [HBG$^+$18] ("robust") and LMS [MCF19] ("simple") [HBD$^+$20, Section 8.1.6].

2.2 Trusted Platform Modules

Trusted Platform Modules (TPMs) are passive co-processors that perform cryptographic operations and store key material in a secure way. A TPM typically contains a small processor that handles the communication protocol, a random number generator, hardware accelerators for cryptography, and a secure non-volatile memory region. The vendors of these devices apply proprietary hardening measures to the hardware structures within the TPM to defeat side-channel and fault-injection attacks. TPMs do not work stand-alone but they always are connected to a host system. Either they are an integral part of the host system or they are connected to the host system via a low level bus system. For example, a TPM can be attached to the host system using the Serial Peripheral Interface (SPI) bus. The TPM 2.0 specification was developed by the Trusted Computing Group (TCG) in 2013 [TCGb] and has since been adopted as ISO/IEC 11889 standard in 2015.

Communication with the host system is realized using a command/response protocol in which the host system issues commands for a variety of tasks and receives responses upon completion of the requested task. For example, the `TPM2_CreateLoaded` command requests the TPM to create a key pair of the specified type and load it into its transient memory. The response in this case includes the created public key as well as an identifier for the transient key-pair object that can be used to perform public and private operations with the key pair. The “Commands“ part of the specification [TCGc] lists all standard TPM commands and their semantics.

TPMs are used to implement a variety of protocols such as Secure Boot, Remote Attestation, and Secure Firmware Updates, where they are the central component for key storage and for performing operations in a way that cannot be influenced by the attached host system. An integrated random number generator allows to create cryptographic keys that are never exported from the TPM. This provides the capability to pin an identity to a specific host system respective TPM. In order to cope with the limited storage resources of a TPM, it is possible to store key material on the host system (encrypted and authenticated) and to load it into the TPM at the time of use.

The TPM 2.0 standard mandates the use of the Rivest-Shamir-Adleman (RSA) public key cryptosystem and optionally that of Elliptic-Curve Cryptography (ECC) for the purposes of digital signatures. These options are however not secure against attackers equipped with sufficiently large quantum computers. Therefore, an integration of post-quantum cryptography into the TPM standard is urgently desired. The FutureTPM project[3] is a European effort to evaluate post-quantum cryptography for the use in TPMs. With the recent mandatory requirement for TPM 2.0 by the Microsoft Windows operating system[4], we can expect an even more widespread adoption of TPMs.

3 Implementation

In this section, we describe the implementation of our streaming interface as well as its integration into our PQC-TPM prototype. Our implementation of the streaming interface is based on the SPHINCS$^+$ reference implementation[5]. We consider only the detached signature API of SPHINCS$^+$, i.e., messages are not included in the signature. Our PQC-TPM prototype is based on the Microsoft TPM 2.0 reference implementation[6].

3.1 Streaming Interface

A SPHINCS$^+$ signature consists of multiple parts as illustrated in Figure 2. First, there is the randomness R followed by a FORS signature. The FORS signature in itself consists of k private key values, each combined with an authentication path in the corresponding Merkle tree. A hypertree signature follows which consists of d Merkle signatures, which each again consist of a WOTS$^+$ signature and an authentication path. A WOTS$^+$ signature is the concatenation of multiple hash-chain nodes that are computed by the WOTS$^+$ chaining function.

Each of the described parts, in turn, consists of one or more n-byte blocks. More precisely, the randomness R, the FORS private key values, every node in the authentication paths, as well as every hash-chain node are all separate byte arrays of size n — because they are each the result of an invocation of the instantiated hash functions that output n bytes. This makes it natural to split signatures into chunks of n bytes as well.

The reference implementation directly accesses a large memory buffer that stores the signature to either write or read signature data. It maintains a pointer to keep track of the current position in the buffer. After a chunk of data has been processed, the pointer is incremented by the corresponding amount of bytes and the next part of the signature is processed.

[3] https://futuretpm.eu
[4] https://www.microsoft.com/en-us/windows/windows-11-specifications#primaryR2
[5] https://github.com/sphincs/sphincsplus
[6] https://github.com/microsoft/ms-tpm-20-ref/

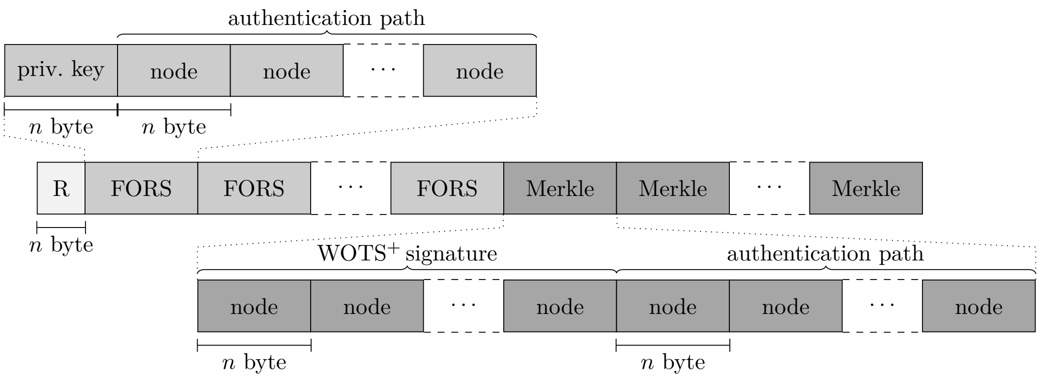

Figure 2: SPHINCS$^+$ signature format.

Our streaming interface can be thought of as an abstraction from a buffer that is *sequentially* written to during signing a message or read from during verifying a signature respectively. Instead of directly accessing the memory at an address inside a buffer, the implementation requests to sequentially read or write one or more blocks of size n from or to the SPHINCS$^+$ signature stream. Since the signature is exactly in the order in which the bytes are processed (for the sign and verify operation alike), this abstraction works. Also, once a chunk of one or more blocks of n bytes has been written, it is not needed for further computations, i.e., it does not need to remain in memory. The same holds for reading: Once the requested chunk has been processed, it is not needed any longer and the chunk does not need to be kept in memory anymore. Only the part of the signature that is currently processed needs to reside in memory.

The streaming API. First, we describe how the integration of our streaming API affects the usage of SPHINCS$^+$. Figure 3 shows the SPHINCS$^+$ API calls for signing and verifying in the reference code. In order to compute a signature, the signature buffer with its length., the message and its length, as well as the secret key are passed into the `crypto_sign_signature` function. For verification, instead of the secret key, the public key is passed into the `crypto_sign_verify` function instead.

Our streaming interface replaces the pointer to the signature (and its length) with a context data structure and requires a few additional functions as shown in Figure 4. Here, the streaming context `streaming_ctx_t` is a data structure that has to be defined depending on the concrete streaming implementation. It is supposed to store all context-related data that is necessary to handle reading from the stream with the `read_stream` function and writing to the stream with the `write_stream` function. The `init_streaming_ctx` and `destroy_streaming_ctx` calls are supposed to handle the initialization and finalization of the stream context. The data structure may also contain buffers for the signature data, e.g., if it is desired to only interact with the I/O layer once a specific amount of data is available for I/O. This can be useful for non-buffered I/O in order to reduce communication overhead resulting from sending and receiving many small messages.

All of the four streaming functions are protocol-dependent and need to be tailored to the specific implementation and usage scenario. Apart from this, no further changes to the application code are required to enable streaming of a signature. The streaming context is passed to the sign and verify functions, respectively, and passed on to subsequent functions to make it available for calls to `read_stream` and `write_stream` as needed. Those two functions are called throughout the SPHINCS$^+$ code instead of accessing a signature buffer directly. Since we removed the the signature buffer from the SPHINCS$^+$ interface, it is now

```
int crypto_sign_signature(uint8_t *sig, size_t *siglen,
                          const uint8_t *m, size_t mlen,
                          const uint8_t *sk);

int crypto_sign_verify(const uint8_t *sig, size_t siglen,
                       const uint8_t *m, size_t mlen,
                       const uint8_t *pk);
```

Figure 3: SPHINCS$^+$ API for detached signatures.

```
int crypto_sign_signature_streaming(streaming_ctx_t *ctx,
                                    const uint8_t *m, size_t mlen,
                                    const uint8_t *sk);

int crypto_sign_verify_streaming(streaming_ctx_t *stream_ctx,
                                 const uint8_t *m, size_t mlen,
                                 const uint8_t *pk);

void init_streaming_ctx(streaming_ctx_t *ctx, ...);

void destroy_streaming_ctx(streaming_ctx_t *ctx);

void write_stream(streaming_ctx_t *ctx,
                  unsigned char *data, unsigned data_len);

void read_stream(streaming_ctx_t *ctx,
                 unsigned char *data, unsigned data_len);
```

Figure 4: Our streaming API.

```
int ret;  // return value

streaming_ctx_t stream_ctx;
init_streaming_ctx(&stream_ctx);      // initalize I/O

/* Sign the message, stream_ctx handles I/O */
ret = crypto_sign_signature_streaming(&stream_ctx, m, mlen, sk);

destroy_streaming_ctx(&stream_ctx);   // finalize I/O
```

Figure 5: Using the streaming API to sign a message `m`.

the responsibility of the streaming abstraction consisting of the functions `read_stream` and `write_stream` to read and write the corresponding data.

Figure 5 shows a typical signing call where the streaming context `stream_ctx`, the message `m`, its length `mlen`, and the secret key `sk` are passed to the streaming version of the signing function `crypto_sign_signature_streaming` by the caller. Verification is done analogously. Aside from providing the streaming context, as well as initializing and finalizing it, there is not much difference compared to using the non-streaming reference code.

Eliminating the signature buffer. The streaming API hides the details of what is buffered at which point. In theory, the data could be read from or written to I/O directly without any buffering (effectively replacing memory operations with data I/O) or the entire signature could be buffered (effectively maintaining the data handling of the reference implementation). In practice, a buffer of a certain size optimal for the underlying I/O interface can be maintained and flushed as data is being written or read.

We first demonstrate that the signature buffer can be reduced to only operating on one n byte chunk of signature data at a time. That means that at no point in time some signature data that has previously been accessed is needed to continue the signature operation, and at no point in time, more than one n byte chunk needs to be accessed.

The following is described from a signer's perspective for simplicity. Although the computations are slightly different for the verifier, it is easy to see that the same general abstraction applies. The components of a SPHINCS$^+$ signature are:

- R: This value is generated by combining the message with a secret PRF key and optional randomness using a hash function. After it has been used to compute the message digest, which is going to be signed by FORS, the value can be written to the stream and removed from memory.
- FORS: Each FORS signature consists of some private keys that are released as part of the signature and an authentication path in a corresponding Merkle tree:
  - Private Keys: A FORS private key is computed by applying the PRF to the SPHINCS$^+$ secret key seed and the appropriate FORS address, according to the SPHINCS$^+$ address scheme. It can be immediately streamed out.
  - Authentication Path: The authentication-path node at a given height is the root of a subtree and can be computed with the tree-hash algorithm directly, given that the index of the left-most leaf in this subtree is specified. In this way it is possible to compute the nodes of the authentication path in order and the buffer for the authentication path can be omitted. To avoid costly recomputations of leaf nodes, the root-node computation is intertwined with the computation of the authentication-path nodes.
  - Public Key: FORS public keys are only an implicit part of the signature, but handling them in an efficient streaming implementation warrants some attention (i.e., no recomputations of nodes, and no large memory overheads). The FORS root nodes are combined to form the public key by computing a tweakable hash over the nodes. This value is then signed by the bottom layer of the SPHINCS$^+$ hypertree. Instead of buffering all root nodes in order to compute the tweakable hash function, we maintain a state of the tweakable hash function and update this state with each newly computed root node, after streaming the corresponding authentication path.
- SPHINCS$^+$ Hypertree: The signature components of the SPHINCS$^+$ hypertree consist of WOTS$^+$ signatures and nodes in the authentication path of corresponding Merkle trees:
  - WOTS$^+$ Signature: The details of the WOTS$^+$ signature generation are described in Section 2.1. Here, we only roughly outline how buffering is avoided. A WOTS$^+$ signature consists of a number of hash-chain nodes that are derived from the private key values and authenticates the tree on the layer below. The computation is typically intertwined with the computation of the WOTS$^+$ public key that is also needed for generating the corresponding leaf in the hypertree. In order to compute the public key, all hash-chain nodes are computed in order. Writing out the signature thus basically only means writing out the correct node while the public key is computed. Finally, for computing the corresponding leaf in the hypertree, the public key values are combined with a tweakable

hash-function call. Instead of buffering the end node of each hash chain and then applying the tweakable hash function, we update the tweakable hash function after each hash-chain end node has been computed.

– Authentication Path: This is analogous to the FORS authentication paths.

Since all elements of the signature can be computed in order, the signature buffer can be reduced to just processing the current chunk of n bytes.

In comparison to this approach, in [HRS16], the authors split a SPHINCS-256 signature into larger chunks. For example, WOTS$^+$ signatures are processed as a single chunk. This does not hinder the computation in 16 kB of RAM though, as the memory is only temporarily required, and the WOTS$^+$ public key is quickly reduced to a single leaf node. Another notable difference is that in [HRS16], the signature format is slightly changed. This is due to a design decision in HORST, the few-time signature scheme that is used for SPHINCS-256 instead of FORS: The top of the tree is cut off at some layer — here layer six — such that only authentication paths up to this layer are computed. This prevents duplications in the top layers, however, it also complicates streaming, since the nodes on the sixth layer are appended at the end of the signature, instead of being part of the respective authentication paths. The solution in [HRS16] is that HORST nodes are tagged with their actual position in the hypertree, rather than producing the nodes in the correct order of the signature. These signature elements are put into the correct order on the receiving end by reordering the nodes accordingly. We want to emphasize that this is not necessary for SPHINCS$^+$ and that the signature format does not need to be changed in our SPHINCS$^+$ streaming implementation.

Details of the implementation. While it is not necessary to rigorously avoid buffering small quantities of n byte chunks, implementing the streaming interface goes hand in hand with reducing the size of the internal buffers that hold parts of the signature. The advantage of working with the granularity of only n bytes — or small multiples of n bytes — is only noticeable when there are no large internal buffers for the signature data. As an example, the reference implementation computes a tweakable hash over a WOTS$^+$ public key by first buffering it entirely. The size of the buffer amounts to up to $67 \times 32 = 2144$ bytes for $n = 32$.

Whenever a tweakable hash function is applied to a buffer with more than two elements, we decided to incrementally update the hash computation instead. This is done for hashing the WOTS$^+$ public key and the FORS root nodes. Instead of applying one single `thash` call for the tweakable hash function, we implement an incremental API with the functions `thash_init`, `thash_update`, and `thash_finalize` for each of the four instantiations SHA-256-simple, SHA-256-robust, SHAKE256-simple, and SHAKE256-robust. As this incremental API stretches the computation over multiple separate calls, a state has to be kept. For the SHAKE256-simple and SHAKE256-robust instantiations, this state amounts to the SHAKE256 state, where for the robust variant, a SHAKE256 state for computing the bitmasks is kept as well. The SHA-256-simple state consists of the SHA-256 state, as well as a buffer to account for the possibility that the current input does not fill an entire block. For SHA-256-robust, an additional state is kept for the Mask Generation Function (MGF) that SPHINCS$^+$ uses for generating masks.

In most places, it is straightforward to change the code to use the stream read and write functions with a granularity of n bytes. Figure 6 shows as an example an excerpt from the WOTS$^+$ leaf-generation reference code. As can be seen in Line 8, the currently computed WOTS$^+$ hash-chain node is copied sequentially to the signature if it is part of the signature. The `memcpy` call can simply be replaced by a `write_stream` call. Likewise, the `pk_buffer` that holds the WOTS$^+$ public key is only needed for the `thash` function call in Line 14. This can be replaced by using the `thash_init`, `thash_update`, and `thash_finalize` functions. Public key nodes are computed sequentially and consumed with a `thash_update` call.

```
for (i=0, buffer=pk_buffer; i<SPX_WOTS_LEN; i++, buffer+=SPX_N) {
  ...
  /* Iterate down the WOTS chain */
  for (k=0;; k++) {
    /* Check if this is the value that needs to be saved as a */
    /* part of the WOTS signature */
    if (k == wots_k) {
      memcpy( info->wots_sig + i * SPX_N, buffer, SPX_N );
    }
    ...
  }
}
/* Do the final thash to generate the public keys */
thash(dest, pk_buffer, SPX_WOTS_LEN, pub_seed, pk_addr);
```

Figure 6: Excerpt from the SPHINCS$^+$ reference code (`wots_gen_leafx1` function).

For the `treehash` function, the reference implementation always generates the tree nodes in the same order. This means that in general, the authentication-path nodes are not computed in the order in which they appear in the signature. Thus, to enable streaming of the authentication paths as well, we modified it to produce the authentication-path nodes in the order in which they are ordered in the signature. This allows us to eliminate the authentication-path buffer as well.

By applying the outlined changes systematically throughout the code, the signature buffer is completely eliminated. While we presented this from the signer's perspective, the same can be done for the verifier. The main difference is that the signer has to traverse each tree with the `treehash` function. For the verifier, all required nodes are provided to directly hash up to the SPHINCS$^+$ root. For each tree, in addition to the provided authentication path, one leaf has to be computed from either the released FORS private key value, the FORS root nodes, or the WOTS$^+$ signature. The WOTS$^+$ and FORS signatures are implicitly verified if the computed root of the hypertree matches the SPHINCS$^+$ public key.

The described streaming implementation does not reduce the storage size, i.e., the size that a signature will require when stored on a medium, such as flash memory. Further, this does not reduce network bandwidth requirements. However, it greatly reduces the memory requirements for the sign and verify operation, since no buffer is required to hold the signature. Furthermore, the key-generation operation also benefits from the changes. While there is no signature buffer that can be eliminated, large intermediate buffers are eliminated, for example to hash the WOTS$^+$ public keys.

General applicability of the streaming implementation. We want to emphasize that in principle our streaming API can be used in every case in which the non-streaming reference API can be used. Neither the public key format nor the signature format are changed (nor the private key format). In general, it is not possible to distinguish between a communication that makes use of the streaming API and the reference implementation, apart from possible timing differences. Since the streaming API is designed as an abstraction from the signature buffer, it can naturally also be used on a signature buffer that resides in memory. In this case, the `stream_write` function merely performs the appropriate `memcpy` calls and keeps track of the currently written bytes and the `stream_read` function simply returns a pointer to the correct location inside the signature buffer.

The streaming API introduces a slight computational overhead, but apart from this and a slightly increased complexity due to implementing and using the streaming context related functions, there is no loss in generality. Moreover, due to the generality, it is easy to integrate our streaming interface into different protocols. The following section demonstrates this on the example of TPMs. As another example, integrating our implementation into a TCP/IP communication can be as simple as calling the respective read and write network-interface functions from inside the `stream_read` and `stream_write` functions. The `streaming_ctx_t` structure would store all necessary context-related data, i.e, typically a file descriptor in socket-APIs. For an established session, the file descriptor can be passed to the `init_streaming_ctx` function. In case the communication needs to be established first, the `init_streaming_ctx` and `destroy_streaming_ctx` calls can open and close the connection.

3.2 TPM Prototype and Streaming Extension

We forked the official TPM 2.0 reference implementation[7] by Microsoft as basis for our prototype TPM with SPHINCS$^+$ signatures and a TPM streaming protocol. The TPM reference implementation can be compiled with different crypto libraries for the required cryptographic primitives, either with the OpenSSL or with the wolfSSL. We chose to use the wolfSSL library, since it is better suited for the use on resource-restricted embedded systems. However, within SPHINCS$^+$ we are using the implementations of the hash functions SHA-256 and SHAKE256 from the SPHINCS$^+$ reference implementation for better comparison with prior art in Section 4.

We needed to apply two major changes to the TPM reference implementation for the integration of SPHINCS$^+$ as signature scheme. The first was to add data structures so that SPHINCS$^+$ can be used for generating keys, as well as for signature generation and verification. The key generation is covered by adding a data structure that stores a SPHINCS$^+$ key pair. We used the data structures that are used for RSA keys as templates for SPHINCS$^+$.

The second major change was the definition of additional commands for signing and verifying using SPHINCS$^+$. The TPM commands `RSA_Encrypt` and `RSA_Decrypt` can be used to perform data signing and signature verification with RSA keys. This is not the only way to use a signature primitive with TPMs, it is however the most direct way to instruct the TPM to sign a message or verify a signature using only one command per operation. We added two commands `PQC_Private` and `PQC_Public` for signing and verification respectively, again using the RSA code as template. We implemented a unit-test on the host for covering key generation, signing, and verification on the TPM using the three commands `TPM_CreateLoaded`, `PQC_Private`, and `PQC_Public`. The command `TPM_CreateLoaded` has a parameter for the selection of the cryptographic scheme that we defined for SPHINCS$^+$. This parameter is used to set the SPHINCS$^+$ parameters during key-pair generation and is stored along-side the key pair on the TPM for use during signing and verification.

In order to send commands to the TPM and to receive its responses, we implemented a variant of the TPM Interface Specification (TIS) protocol [TCGa]. TIS defines a 24 bit address space that is mapped to control registers and data buffers inside the TPM. Read and write operations on specific addresses are used to transfer commands and data to the TPM, to receive responses from it, and to initiate the execution of commands.

The specification describes a simple SPI-based protocol for implementing read and write operations for TPMs that are not directly connected to a memory bus. Messages in this SPI protocol start with a byte that uses the first bit to indicate whether a read or write operation is performed and the remaining 7 bit to define the length of the data in bytes. This initial byte is followed by a 24 bit address indicating the source (for a read operation) or destination (for a write operation). After this, data is transferred.

[7]https://github.com/microsoft/ms-tpm-20-ref/

The host system initiates all communication with the TPM and polls a 32 bit status register using the SPI-based protocol to detect if the TPM is ready to receive commands or if a response is ready. We only implemented a minimal subset of the address space that is defined in the TIS specification. The two most important addresses are the 32 bit status register `STS` at address `000018h` and the 32 bit FIFO command-and-response register `DATA_FIFO` at address `000024h`. Specific bits of the `STS` are used by the TPM to communicate its readiness to receive commands or transmit responses as well as for the host system to initiate the execution of a previously written TPM command.

We propose an extension to this communication protocol for the streaming of data between the host and the TPM. We extend the TIS interface by two addresses, the 32 bit `IOSTREAM` FIFO register at address `000030h` for the streaming of data and the 32 bit `STREAMSIZE` register at address `000040h` to communicate the size of the data to be streamed. The `STS` register is polled by the host platform regularly. Several of its bits are marked as reserved by the TIS specification. We decided to use two of those reserved bits to signal that the TPM is ready to send (`STS` bit 23) or receive (`STS` bit 24) data. If one of these bits is set when the host system reads the `STS` register, it reads out the `STREAMSIZE` register, which holds the number of bytes that the TPM is able to send or receive. The data is then read from or written to the `IOSTREAM` register, which acts as a FIFO. This helps to reduce additional polling during streaming, since the host system is already polling the `STS` register during the execution of any TPM command to detect if a response is ready.

An alternative would be to formalize the sequence of streaming messages in a state machine and to issue special commands asking for further data or to transmit data via the `DATA_FIFO` register. This approach however would introduce a significant overhead in transmitted data and all data would have to pass through the serialization and deserializing layers of the TPM firmware. Furthermore, the implementation of a cryptographic scheme using such a state-machine-based communication layer would require significant changes to the program flow into a state-machine as well instead of being able to handle I/O transparently during the cryptographic computations as described in Section 3.1 for SPHINCS$^+$.

We used the SPI controller (in the SPI "slave" role) of the STM32F4 SoC to implement the TIS protocol. Our implementation is driven by an interrupt service routine that is triggered every time a byte is received on the SPI bus. The interrupt operates a small state machine that handles writing to and reading from the available addresses, including the streaming of signature data. This approach results in an interruption of the code for every byte that is transferred. These frequent interrupts could be avoided using a dedicated hardware implementation of the TIS protocol on an actual TPM.

4 Evaluation

Each of the parameter sets in Table 1 can be instantiated with different choices for the hash functions. These variants determine the implementation of the tweakable hash function and also what has to be maintained in the hash states. This directly influences the considered metrics, i.e., cycle count and stack usage. Therefore, we give separate measurements for the different instantiations. We consider the four variants SHA-256-simple, SHA-256-robust, SHAKE256-simple and SHAKE256-robust.

We built the binaries with GCC 9.2.1 and the "–O2" flag which seems to offer a good trade-off between the different metrics. We are evaluating our implementation on a STM32F4 Discovery board with an ARM Cortex-M4 CPU. The board offers several choices for clock frequencies. We are using the default frequency of 168 MHz if not explicitly stated differently. For our measurements with the TPM integration we used a Raspberry Pi 4 Model B in the role of the host system, because it provides an SPI controller for communication with our TPM prototype.

Table 2: Relative performance of the streaming modifications compared to the reference implementation and the pqm4/PQClean implementation.

	reference impl.			pqm4/PQClean		
Parameter Set	Keygen	Sign	Verify	Keygen	Sign	Verify
sphincs-sha256-128f-robust	100.8 %	100.8 %	101.2 %	117.9 %	111.3 %	123.4 %
sphincs-sha256-128s-robust	97.5 %	97.6 %	96.1 %	117.9 %	117.8 %	123.0 %
sphincs-sha256-192f-robust	100.4 %	100.4 %	101.2 %	119.1 %	113.1 %	120.2 %
sphincs-sha256-192s-robust	100.4 %	100.5 %	100.0 %	119.2 %	118.7 %	121.4 %
sphincs-sha256-256f-robust	100.0 %	100.1 %	100.0 %	118.5 %	115.4 %	120.1 %
sphincs-sha256-256s-robust	99.8 %	99.9 %	98.3 %	118.7 %	118.1 %	117.8 %
sphincs-sha256-128f-simple	102.3 %	102.3 %	107.9 %	110.1 %	103.8 %	114.6 %
sphincs-sha256-128s-simple	98.9 %	99.0 %	105.1 %	110.0 %	110.0 %	114.6 %
sphincs-sha256-192f-simple	101.6 %	101.7 %	100.4 %	110.3 %	104.7 %	112.7 %
sphincs-sha256-192s-simple	101.9 %	101.9 %	104.1 %	110.4 %	110.1 %	114.7 %
sphincs-sha256-256f-simple	100.9 %	100.9 %	98.7 %	110.9 %	107.9 %	110.9 %
sphincs-sha256-256s-simple	100.5 %	100.6 %	93.6 %	111.3 %	110.8 %	106.8 %
sphincs-shake256-128f-robust	100.4 %	100.4 %	101.8 %	261.8 %	247.3 %	262.0 %
sphincs-shake256-128s-robust	94.2 %	94.2 %	96.3 %	261.8 %	261.5 %	254.3 %
sphincs-shake256-192f-robust	100.4 %	100.4 %	99.1 %	260.4 %	246.7 %	258.7 %
sphincs-shake256-192s-robust	100.4 %	100.4 %	97.3 %	260.4 %	259.9 %	256.8 %
sphincs-shake256-256f-robust	100.3 %	100.3 %	94.3 %	259.0 %	251.9 %	252.4 %
sphincs-shake256-256s-robust	100.6 %	100.6 %	99.1 %	259.1 %	258.1 %	261.7 %
sphincs-shake256-128f-simple	100.5 %	100.5 %	96.7 %	257.0 %	242.6 %	252.0 %
sphincs-shake256-128s-simple	94.2 %	94.2 %	102.6 %	257.0 %	256.7 %	255.1 %
sphincs-shake256-192f-simple	100.4 %	100.4 %	98.9 %	255.0 %	241.6 %	253.2 %
sphincs-shake256-192s-simple	100.4 %	100.4 %	103.4 %	255.1 %	254.7 %	261.7 %
sphincs-shake256-256f-simple	100.2 %	100.2 %	101.5 %	253.2 %	246.3 %	250.8 %
sphincs-shake256-256s-simple	100.4 %	100.4 %	95.8 %	253.4 %	252.6 %	256.3 %

4.1 Streaming Interface

The mupq project provides implementations and optimizations for several PQC schemes for several embedded platforms. It's subproject pqm4[8] is dedicated to the Cortex-M4 platform using the STM32F4 Discovery board. For SPHINCS$^+$, it uses the implementation from the PQClean project[9], which is based on the SPHINCS$^+$ reference implementation.

In order to verify that our streaming interface does not introduce a significant performance penalty, we compare the cycle counts and the stack sizes of the selected SPHINCS$^+$ parameter sets with the reference implantation and the mupq/PQClean implementation of SPHINCS$^+$ on the Cortex-M4 platform. The mupq project provides performance measurements that are frequently updated. We compare the performance of our streaming interface to the measurements in the file "benchmarks.csv" from git commit 12d5e56.

Performance. Table 2 shows the relative runtime of the streaming interface compared to the reference and to the pqm4/PQClean implementations. The pqm4 project is running the STM32F4 Discovery board at only 24 MHz in order to avoid memory-access effects when reading instructions from the slow flash memory. Therefore, we performed the measurements of our streaming interface at the frequencies 168 MHz for comparison to the reference implementation and at 24 MHz for comparison to mupq and report the respective relative performance to the reference and the pqm4/PQClean implementations in percent (detailed cycle counts are listed in Table 5 in the Appendix).

[8] https://github.com/mupq/pqm4
[9] https://github.com/PQClean/PQClean

For the reference implementation the difference in the cycle count to our streaming implementation varies between one or two percent for key generation and signing. Besides the integration of the streaming interface, our code also replaces the compile-time selection of the parameter set in the reference code with a runtime-selection, introducing some `if-then` clauses, which might be the cause of some of the cycle-count differences. There are two outliers for the SHAKE256-128s parameter sets (both robust and simple) in the comparison with the reference implementation where our implementation requires only 94.2 % of the time of the reference implementation. However, overall our modifications to the control flow of the reference implementation when integrating the streaming interface do not have a major impact on the performance.

The difference to pqm4/PQClean is more pronounced: The pqm4/PQClean performance for key generation and signing is 10 % to 20 % faster than our version for the SHA-256 variants and even about 2.5 times faster for the SHAKE256 variants. This is due to the fact that the pqm4 project is using optimized SHA-256 and SHAKE256 code for the Cortex-M4 CPU. We did not incorporate these optimizations in order to remain closer to the reference implementation. On an actual PQC-TPM, we would recommend to use hardware acceleration for all hash computations to achieve a significant improvement over both our and the pqm4/PQClean code.

The runtime varies more significantly for the verification operation — since the verification time itself varies for different inputs (as opposed to key generation and signing which have a runtime that is quite independent from inputs). However, the performance differences for verification are qualitatively similar to those of key generation and signing.

Stack size. For stack measurements we are using an implementation of the streaming interface that is simply writing into a large buffer similar to the reference implementation. This signature buffer is not part of the stack-size measurements for all, the reference, the pqm4/PQClean, and our streaming implementation.

Table 3 shows the stack sizes of the reference implementation (columns "ref.") and the pqm4/PQClean implementation (columns "pqm4") compared to our streaming variant (columns "strm."). We also provide the signature sizes to emphasise that both the reference and the pqm4/PQClean implementations require a large data buffer for storing the signature in addition to the stack requirements.

Due to the modifications to the data handling on the stack and in buffers, we are reducing the stack size significantly even if the buffer for storing the complete signature required by the reference and the pqm4/PQClean implementations is not taken into account. The buffer for our streaming variant can be entirely avoided if all data is transferred or consumed right away or it can be any size up to the size of an entire signature. For the TPM example, we are using a 1 kB buffer (see Section 4.2).

The stack requirements of pqm4/PQClean are significantly lower than those of the reference implementation. However, our streaming implementation requires even less stack than pqm4/PQClean — except for a few corner cases for the SHAKE256 parameter sets with $n = 16$. Our implementation requires only 1 296 kB to 2 976 kB of stack, which is between 23 % to 73 % of the stack size of the reference implementation; the difference is most significant for the higher security levels using both SHA-256 and SHAKE256. Compared to pqm4/PQClean, in some cases we achieve a reduction in stack usage to 33%.

Comparison to prior streaming approaches. Our implementation cannot directly be compared to the work by Hülsing et al. in [HRS16], since they are investigating signature streaming for the predecessor SPHINCS-256 of SPHINCS$^+$. SPHINCS-256 uses different performance parameters and a different few-time signature scheme than SPHINCS$^+$. Their primary goal is to fit into the 16 kB RAM of their Cortex-M3 CPU. They require much fewer cycles for key generation but many more cycles for signing with their SPHINCS-256

Table 3: Stack sizes for the SPHINCS$^+$ reference implementation (“ref”), the pqm4/PQClean implementation (“pqm4”), and our streaming interface implementation (“strm.”) as well as the corresponding signature sizes in bytes.

Parameter Set (sphincs-. . .)	Keygen			Sign			Verify			Sig.
	ref.	pqm4	strm.	ref.	pqm4	strm.	ref.	pqm4	strm.	
sha256-128f-robust	3 688	2 256	1 960	3 176	2 320	2 000	3 344	2 808	1 728	17 088
sha256-128s-robust	3 984	2 472	2 056	3 264	2 544	2 088	2 592	2 112	1 656	7 856
sha256-192f-robust	6 536	3 680	2 192	5 336	3 832	2 264	4 848	4 040	1 856	35 664
sha256-192s-robust	6 928	4 104	2 336	5 472	3 992	2 360	4 728	3 376	1 888	16 224
sha256-256f-robust	10 456	5 792	2 456	8 272	5 760	2 512	7 424	5 656	2 088	49 856
sha256-256s-robust	10 816	6 064	2 584	8 400	5 904	2 616	7 424	5 360	1 960	29 792
sha256-128f-simple	2 904	2 104	1 632	2 392	2 168	1 688	2 592	2 656	1 384	17 088
sha256-128s-simple	3 096	2 432	1 736	2 480	2 392	1 768	1 904	1 960	1 296	7 856
sha256-192f-simple	5 072	3 520	1 840	3 872	3 560	1 896	3 816	3 880	1 480	35 664
sha256-192s-simple	5 464	3 944	1 992	4 008	3 832	2 016	3 160	3 216	1 456	16 224
sha256-256f-simple	8 160	5 512	2 080	5 872	5 592	2 104	5 424	5 488	1 876	49 856
sha256-256s-simple	8 416	5 896	2 216	6 000	5 736	2 232	5 024	5 080	1 868	29 792
shake256-128f-robust	4 052	2 012	2 336	3 544	2 176	2 392	3 708	2 556	2 208	17 088
shake256-128s-robust	4 352	2 336	2 432	3 632	2 288	2 464	2 956	1 860	2 120	7 856
shake256-192f-robust	6 892	3 436	2 560	5 696	3 576	2 632	5 204	3 788	2 328	35 664
shake256-192s-robust	7 288	3 856	2 704	5 832	3 736	2 728	4 980	3 124	2 176	16 224
shake256-256f-robust	10 912	5 436	2 816	8 624	5 504	2 872	7 880	5 404	2 448	49 856
shake256-256s-robust	11 168	5 816	2 944	8 752	5 648	2 976	7 772	4 996	2 320	29 792
shake256-128f-simple	3 476	2 012	2 104	2 968	2 068	2 144	3 164	2 556	1 960	17 088
shake256-128s-simple	3 776	2 336	2 200	3 056	2 288	2 232	2 468	1 860	1 800	7 856
shake256-192f-simple	5 660	3 436	2 320	4 464	3 468	2 368	4 404	3 788	1 956	35 664
shake256-192s-simple	6 056	3 856	2 464	4 600	3 736	2 488	3 748	3 124	1 936	16 224
shake256-256f-simple	8 644	5 436	2 568	6 464	5 504	2 592	6 012	5 404	2 072	49 856
shake256-256s-simple	9 008	5 816	2 696	6 592	5 648	2 712	5 612	4 996	2 112	29 792

parameter set compared to us using the SPHINCS$^+$ parameter sets. Their stack usage is between 6 kB and 9 kB depending on compiler parameters.

Gonzalez et al. are providing performance and stack measurements for two SPHINCS$^+$ parameter sets for verifying a streamed signature in [GHK$^+$21] on a Cortex-M3 platform with 8 kB of memory. Their cycle counts are not comparable to our implementation since we are using a different CPU for our measurements but their stack size is similar to our version. However, they are splitting the signature in relatively large parts of up to 5 kB along the layers of the SPHINCS$^+$ hypertree, while our implementation is more flexible in regard to the streaming interface and buffer sizes and independent of the SPHINCS$^+$ hypertree structure.

4.2 TPM Integration

Table 4 shows performance measurements of the SPHINCS$^+$ variants integrated into our prototype TPM implementation. We compute the I/O overhead (column “Embedded – I/O”) as the relative difference between the cycle counts from the measurements of the stand-alone streaming implementation (column “Embedded – strm.”) and the cycle count of the corresponding operation in the TPM integration (column “Embedded – TPM”) as “`(TPM - strm.) / (TPM * 100)`”. We also report the overall wall clock time of the corresponding TPM operation (including TPM-protocol overhead etc.) as seen from the host in seconds (column “Host – total”).

Table 4: Performance data for key generation, signing, and verification on the embedded device and on the host. For the embedded device, we list the cycle counts of the reference implementation including the streaming interface as "strm." in mega cycles, the integration of SPHINCS$^+$ into the TPM prototype as "TPM" in mega cycles, and the communication overhead, i.e., the difference between the two in percent, as "I/O Δ". For the host, we list the overall wall-clock time from issuing a TPM command until its completion (including I/O) as "total" in seconds.

	Key Generation				Signing				Verification			
	Embedded			Host	Embedded			Host	Embedded			Host
Parameter Set	strm. [mcyc]	TPM [mcyc]	I/O Δ	total [s]	strm. [mcyc]	TPM [mcyc]	I/O Δ	total [s]	strm. [mcyc]	TPM [mcyc]	I/O Δ	total [s]
sphincs-sha256-128f-robust	55.0	60.5	9.1 %	0.531	1 280	1 960	34.9 %	11.8	82.7	470	82.4 %	2.90
sphincs-sha256-128s-robust	3 520	3 870	9.1 %	23.3	26 400	29 500	10.3 %	176	27.8	206	86.5 %	1.33
sphincs-sha256-192f-robust	81.8	91.3	10.4 %	0.727	2 160	3 580	39.8 %	21.4	123	928	86.8 %	5.63
sphincs-sha256-192s-robust	5 240	5 840	10.4 %	35.0	48 400	54 600	11.4 %	326	44.3	406	89.1 %	2.52
sphincs-sha256-256f-robust	300	330	9.0 %	2.17	6 120	8 340	26.7 %	49.8	177	1 320	86.6 %	7.98
sphincs-sha256-256s-robust	4 800	5 280	9.0 %	31.6	58 900	65 800	10.5 %	392	90.0	767	88.3 %	4.67
sphincs-sha256-128f-simple	27.3	29.7	8.2 %	0.340	640	1 250	48.9 %	7.54	39.8	420	90.5 %	2.61
sphincs-sha256-128s-simple	1 750	1 900	8.2 %	11.5	13 300	14 800	10.2 %	88.2	13.6	189	92.8 %	1.23
sphincs-sha256-192f-simple	40.2	44.4	9.6 %	0.447	1 080	2 360	54.3 %	14.1	58.4	850	93.1 %	5.17
sphincs-sha256-192s-simple	2 570	2 840	9.6 %	17.1	24 400	27 700	11.6 %	165	21.2	386	94.5 %	2.40
sphincs-sha256-256f-simple	106	120	10.9 %	0.914	2 230	4 110	45.8 %	24.6	60.6	1 180	94.9 %	7.14
sphincs-sha256-256s-simple	1 700	1 910	10.9 %	11.6	22 000	25 800	14.7 %	154	28.9	689	95.8 %	4.21
sphincs-shake256-128f-robust	410	450	8.9 %	2.85	9 520	11 000	13.5 %	65.7	580	1 030	43.7 %	6.25
sphincs-shake256-128s-robust	26 300	28 800	8.9 %	172	197 000	217 000	9.1 %	1 290	202	389	48.1 %	2.42
sphincs-shake256-192f-robust	601	660	8.9 %	4.12	15 200	17 900	14.9 %	107	849	1 750	51.6 %	10.6
sphincs-shake256-192s-robust	38 500	42 200	8.9 %	252	335 000	369 000	9.2 %	2 200	294	689	57.3 %	4.21
sphincs-shake256-256f-robust	1 590	1 750	9.0 %	10.6	31 200	36 000	13.2 %	214	855	2 060	58.4 %	12.4
sphincs-shake256-256s-robust	25 400	27 900	9.0 %	167	288 000	318 000	9.4 %	1 900	429	1 120	61.7 %	6.77
sphincs-shake256-128f-simple	212	234	9.3 %	1.56	4 970	6 030	17.6 %	36.0	298	698	57.3 %	4.26
sphincs-shake256-128s-simple	13 600	15 000	9.3 %	89.5	103 000	114 000	9.5 %	680	104	286	63.6 %	1.81
sphincs-shake256-192f-simple	311	344	9.5 %	2.23	8 040	10 100	20.0 %	60.0	429	1 250	65.7 %	7.54
sphincs-shake256-192s-simple	19 900	22 000	9.5 %	131	179 000	198 000	9.8 %	1 180	152	517	70.6 %	3.18
sphincs-shake256-256f-simple	823	910	9.7 %	5.63	16 500	19 900	17.0 %	119	438	1 580	72.2 %	9.50
sphincs-shake256-256s-simple	13 200	14 600	9.7 %	87.0	156 000	174 000	10.3 %	1 040	214	883	75.8 %	5.36

The modified reference implementation of the TPM firmware performs serialization and deserialization, formatting of commands, and parsing the responses of the TPM. On the host system we measured "wall-clock" time with a resolution of milliseconds for the complete execution of three commands, including transmission of the command, reception of the response, and streaming I/O that occurs in between.

I/O overhead. Since the prototype TPM is connected to the host via SPI, the throughput of the communication is rather low. In addition, since the development board that is running the TPM code does not have dedicated hardware for SPI communication, all SPI interrupts and data transfers need to be handled by the CPU. The speed of the SPI communication can be controlled by the SPI host. Therefore, we gradually increased the SPI bus speed on the Raspberry Pi to find the maximum speed without transmission errors and conducted the experiments at the resulting speed of 4 MHz.

For key generation, there is an about 10 % overhead caused by the general host-TPM communication and the SPI interrupt handling due to frequent polling from the host on the SPI bus, despite the fact that there is no data I/O during the computation of the keys. For signing, the I/O overhead is large for the "f" parameter sets due to their large signature sizes. The effect is quite significant for the SHA-256 parameter sets compared to the SHAKE256 variants since the computational cost of SHAKE256 is higher for our implementation.

For verification, the communication overhead is even more pronounced. In case of SHA-256, communication takes well over 80 % of the time; also for the SHAKE256 variants, the communication overhead is around or more than half. Therefore, the performance advantage of the "s" over the "f" parameter sets for verification is rather dominated by the communication time of the around two times smaller signatures than by the up to three times faster verification time.

Wall-clock time. The key-generation time using the TPM command `TPM_CreateLoaded` for the fast "f" parameter sets overall is relatively low with under 3 s for the SHA-256 variants and under 12 s for the SHAKE256 variants. The key-generation time for the slower size-optimized "s" parameter sets is one order of magnitude slower. Signing time using the command `PQC_Private` is quite high for SHA-256 and extremely high for some SHAKE256 variants. Verification time is relatively low for all parameter sets but in particular for the "s" parameter sets.

As mentioned in Section 2.1, most of the time in all SPHINCS$^+$ operations is spent in the hash-functions. Therefore, providing optimized implementations or hardware acceleration for SHA-256 and SHAKE256 in a PQC-TPM will significantly speed up the wall-clock time for SPHINCS$^+$ operations. In our experiments, verification is dominated by the communication time and also for signing using the "f" parameters the communication time is significant. However, communication time can be reduced as well with dedicated hardware for SPI or when the TPM is tightly coupled with the host system.

The "f" parameter sets are intended to provide faster key generation and signing time at the cost of larger signatures and slower verification. This effect is clearly visible in our TPM-prototype measurements. Therefore, we recommend the "s" parameter sets for applications where a TPM verifies SPHINCS$^+$ signatures and the "f" parameter sets when a TPM is the owner of a SPHINCS$^+$ private key (i.e., performs key generation and signing). If a TPM is required to perform all the operations key generation, signing, and verification, then the "f" parameter sets may overall be the better choice. Our streaming interface enables even TPMs and embedded devices with very restricted memory resources to handle the larger "f" signatures.

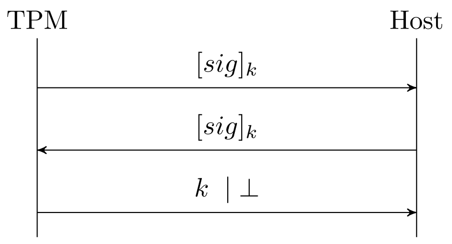

Figure 7: Example for fault countermeasure for streamed signatures.

5 Fault Attacks

Kannwischer et al. describe a random fault attack on hash-based signature schemes including SPHINCS$^+$ in [GKP$^+$18]. Their fault attack operates by causing a random fault in the second highest layer of the hypertree. This randomizes the corresponding tree root at that layer, which is subsequently signed using the WOTS$^+$ scheme. Repeating the fault injection several times leads to a number of faulty signatures for "different random messages" (i.e., different root nodes on the second highest layer) that are getting signed by the same WOTS$^+$ signing key. If the attacker can obtain a large enough number of faulty signatures, sufficient information on the secret WOTS$^+$ signing key can be recovered, which effectively reveals an equivalent signing key. Hence, the attacker can sign arbitrary messages under an equivalent private key.

The streaming method described in this paper for the signature computation is particularly susceptible to random fault attacks, because the communication during the signature computation clearly signals the current part of the signature that is being computed. An attacker can use this to facilitate triggering the random fault at the correct hypertree layer.

The general countermeasure against this kind of random fault attacks on signature schemes is to store and verify the signature after its computation before releasing it to the communication channel. This, however, is impossible in a streaming scenario — since the main purpose of the signature streaming is to avoid storing the entire signature.

To protect against a random fault attack in the TPM signature-streaming scenario, we propose the following solution: The TPM encrypts the signature stream with an ephemeral symmetric key. Once the entire signature has been generated, the host streams the encrypted signature back to the TPM. The TPM decrypts the signature stream and verifies the signature (again using our streaming API without storing the entire signature). If and only if the signature is correctly verified, the TPM then sends the symmetric key to the host, who then can decrypt and further process the signature. If the TPM cannot verify the signature, the symmetric key is not transmitted and discarded. Figure 7 shows a diagram for this protocol. This solution can be transferred to other signature-streaming scenarios where an encrypted signature can be streamed back to the signer.

The additional cost for this countermeasure is that every signature computation requires additional communication overhead for streaming the encrypted signature back to the signer and additional computational cost for encrypting the signature as well as for the signature verification. However, since signature verification is relatively cheap compared to the signature computation, this overhead can be relatively small (see Table 4). This countermeasure forces the attacker to inject the exact same fault during verification as during the signature generation, which renders random fault attacks impossible and increases the cost for fault attacks significantly. Therefore, SPHINCS$^+$ signatures can be streamed securely into and out of resource-restricted embedded devices like TPMs and Smart-Cards that do not have sufficient memory to store an entire signature.

Acknowledgements

This work has been funded by the German Federal Ministry of Education and Research (BMBF) under the project "QuantumRISC" (IDs 16KIS1033K and 16KIS1037).

References

[ACZ18] D. Amiet, A. Curiger, and P. Zbinden. FPGA-based accelerator for SPHINCS-256. *IACR TCHES*, 2018(1):18–39, 2018. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i1.18-39. https://tches.iacr.org/index.php/TCHES/article/view/831.

[BDH11] J. A. Buchmann, E. Dahmen, and A. Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In B.-Y. Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 117–129. Springer, Heidelberg, 2011. DOI: 10.1007/978-3-642-25405-5_8.

[BHH+15] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 368–397. Springer, Heidelberg, April 2015. DOI: 10.1007/978-3-662-46800-5_15.

[BHK+19] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe. The SPHINCS$^+$ signature framework. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 2129–2146. ACM Press, November 2019. DOI: 10.1145/3319535.3363229.

[CKR+20] F. Campos, T. Kohlstadt, S. Reith, and M. Stöttinger. LMS vs XMSS: comparison of stateful hash-based signature schemes on ARM Cortex-M4. In A. Nitaj and A. M. Youssef, editors, *AFRICACRYPT 20*, volume 12174 of *LNCS*, pages 258–277. Springer, Heidelberg, July 2020. DOI: 10.1007/978-3-030-51938-4_13.

[GHK+21] R. Gonzalez, A. Hülsing, M. J. Kannwischer, J. Krämer, T. Lange, M. Stöttinger, E. Waitz, T. Wiggers, and B. Yang. Verifying post-quantum signatures in 8 kB of RAM. Cryptology ePrint Archive, Report 2021/662 (accepted at PQCrypto 2021), 2021. https://eprint.iacr.org/2021/662.

[GKP+18] A. Genêt, M. J. Kannwischer, H. Pelletier, and A. McLauchlan. Practical fault injection attacks on SPHINCS. Cryptology ePrint Archive, Report 2018/674, 2018. https://eprint.iacr.org/2018/674.

[HBD+20] A. Hülsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kolbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens. SPHINCS+. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[HBG+18] A. Hülsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. XMSS: extended merkle signature scheme. *RFC*, 8391:1–74, 2018. DOI: 10.17487/RFC8391. URL: https://doi.org/10.17487/RFC8391.

[HRS16] A. Hülsing, J. Rijneveld, and P. Schwabe. ARMed SPHINCS - computing a 41 KB signature in 16 KB of RAM. In C.-M. Cheng, K.-M. Chung, G. Persiano, and B.-Y. Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 446–470. Springer, Heidelberg, March 2016. DOI: 10.1007/978-3-662-49384-7_17.

[Köl18] S. Kölbl. Putting wings on SPHINCS. In T. Lange and R. Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 205–226. Springer, Heidelberg, 2018. DOI: 10.1007/978-3-319-79063-3_10.

[KRS$^+$] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. PQM4: post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KRS$^+$19] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. pqm4: testing and benchmarking NIST PQC on ARM cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844.

[MAA$^+$20] D. Moody, G. Alagic, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and J. Alperin-Sheriff. Status report on the second round of the NIST post-quantum cryptography standardization process, 2020. DOI: https://doi.org/10.6028/NIST.IR.8309.

[MCF19] D. McGrew, M. Curcio, and S. Fluhrer. Hash-based signatures. IETF Crypto Forum Research Group, 2019. https://datatracker.ietf.org/doc/html/draft-mcgrew-hash-sigs-15.

[Mer90] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238. Springer, Heidelberg, August 1990. DOI: 10.1007/0-387-34805-0_21.

[RKK20] J. Roth, E. G. Karatsiolis, and J. Krämer. Classic McEliece implementation with low memory footprint. In P. Liardet and N. Mentens, editors, *CARDIS 2020*, volume 12609 of *LNCS*, pages 34–49. Springer, Heidelberg, November 2020. DOI: 10.1007/978-3-030-68487-7.

[TCGa] Trusted Computing Group. TCG PC client specific TPM interface specification (TIS). *Specification Version 1.3*, 1, 2013. https://trustedcomputinggroup.org/resource/pc-client-platform-tpm-profile-ptp-specification/.

[TCGb] Trusted Computing Group. Trusted Platform Module Library, Part 1: Architecture. *Specification version: 1.59 Family 2.0*, 1, 2019. https://trustedcomputinggroup.org/resource/tpm-library/.

[TCGc] Trusted Computing Group. Trusted Platform Module Library, Part 3: Commands. *Specification version: 1.59 Family 2.0*, 1, 2019. https://trustedcomputinggroup.org/resource/tpm-library/.

[WJW$^+$19] W. Wang, B. Jungk, J. Wälde, S. Deng, N. Gupta, J. Szefer, and R. Niederhagen. XMSS and embedded systems. In K. G. Paterson and D. Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 523–550. Springer, Heidelberg, August 2019. DOI: 10.1007/978-3-030-38471-5_21.

Appendix

Table 5: Cycle counts (in mega cycles) for key generation, signing, and verification for the reference implementation and the streaming API at 168 MHz as well as for the pqm4/pqclean implementation the streaming API at 24 MHz rounded to three significant figures.

	stream vs. reference impl. (168 MHz)						stream vs. pqm4/pqclean (24 MHz)					
	Keygen		Sign		Verify		Keygen		Sign		Verify	
Parameter Set	ref.	strm.	ref.	strm.	ref.	strm.	pqm4	strm.	pqm4	strm.	pqm4	strm.
sphincs-sha256-128f-robust	54.6	55.0	1 270	1 280	81.7	82.7	30.5	36.0	750	835	43.9	54.2
sphincs-sha256-128s-robust	3 610	3 520	27 100	26 400	28.9	27.8	1 950	2 300	14 700	17 300	14.8	18.2
sphincs-sha256-192f-robust	81.5	81.8	2 150	2 160	121	123	45.2	53.8	1 250	1 420	67.1	80.6
sphincs-sha256-192s-robust	5 220	5 240	48 100	48 400	44.3	44.3	2 890	3 440	26 800	31 700	23.9	29.1
sphincs-sha256-256f-robust	300	300	6 110	6 120	177	177	165	195	3 450	3 980	95.8	115
sphincs-sha256-256s-robust	4 810	4 800	58 900	58 900	91.5	90.0	2 630	3 130	32 400	38 200	49.6	58.5
sphincs-sha256-128f-simple	26.7	27.3	625	640	36.9	39.8	16.1	17.7	400	416	22.6	25.8
sphincs-sha256-128s-simple	1 770	1 750	13 400	13 300	12.9	13.6	1 030	1 140	7 850	8 630	7.71	8.84
sphincs-sha256-192f-simple	39.5	40.2	1 060	1 080	58.1	58.4	23.7	26.2	669	701	33.6	37.9
sphincs-sha256-192s-simple	2 520	2 570	24 000	24 400	20.4	21.2	1 520	1 670	14 400	15 800	12.0	13.8
sphincs-sha256-256f-simple	106	106	2 210	2 230	61.4	60.6	62.6	69.4	1 340	1 450	35.5	39.4
sphincs-sha256-256s-simple	1 700	1 700	21 900	22 000	30.9	28.9	998	1 110	12 900	14 300	17.6	18.7
sphincs-shake256-128f-robust	409	410	9 480	9 520	570	580	113	297	2 790	6 890	160	420
sphincs-shake256-128s-robust	27 900	26 300	209 000	197 000	210	202	7 260	19 000	54 600	143 000	57.5	146
sphincs-shake256-192f-robust	598	601	15 100	15 200	856	849	167	435	4 460	11 000	238	615
sphincs-shake256-192s-robust	38 300	38 500	333 000	335 000	302	294	10 700	27 900	93 300	242 000	83.0	213
sphincs-shake256-256f-robust	1 580	1 590	31 100	31 200	907	855	445	1 150	8 990	22 700	246	620
sphincs-shake256-256s-robust	25 300	25 400	287 000	288 000	433	429	7 120	18 400	81 000	209 000	119	311
sphincs-shake256-128f-simple	211	212	4 950	4 970	308	298	59.8	154	1 480	3 590	85.4	215
sphincs-shake256-128s-simple	14 400	13 600	110 000	103 000	102	104	3 830	9 830	29 100	74 700	29.6	75.4
sphincs-shake256-192f-simple	310	311	8 010	8 040	433	429	88.3	225	2 410	5 820	123	310
sphincs-shake256-192s-simple	19 800	19 900	178 000	179 000	147	152	5 650	14 400	50 800	129 000	42.1	110
sphincs-shake256-256f-simple	821	823	16 500	16 500	431	438	235	596	4 860	12 000	127	317
sphincs-shake256-256s-simple	13 100	13 200	156 000	156 000	223	214	3 760	9 540	44 900	113 000	60.5	155