Verifying Post-Quantum Signatures in 8 kB of RAM

Ruben Gonzalez[1], Andreas Hülsing[2], Matthias J. Kannwischer[3], Juliane Krämer[4], Tanja Lange[2], Marc Stöttinger[5], Elisabeth Waitz[6], Thom Wiggers[7], and Bo-Yin Yang[8]

[1] Hochschule Bonn-Rhein-Sieg, Bonn, Germany
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
[3] Max Planck Institute for Security and Privacy, Bochum, Germany
[4] Technische Universität Darmstadt, Darmstadt, Germany
[5] Hessen3C, Wiesbaden, Germany
[6] Elektrobit Automotive GmbH, Erlangen, Germany
[7] Radboud University, Nijmegen, The Netherlands
[8] Academica Sinica, Taipei, Taiwan
`streaming-pq-sigs@kannwischer.eu`

Abstract. In this paper, we study implementations of post-quantum signature schemes on resource-constrained devices. We focus on verification of signatures and cover NIST PQC round-3 candidates Dilithium, Falcon, Rainbow, GeMSS, and SPHINCS$^+$. We assume an ARM Cortex-M3 with 8 kB of memory and 8 kB of flash for code; a practical and widely deployed setup in, for example, the automotive sector. This amount of memory is insufficient for most schemes. Rainbow and GeMSS public keys are too big; SPHINCS$^+$ signatures do not fit in this memory. To make signature verification work for these schemes, we stream in public keys and signatures. Due to the memory requirements for efficient Dilithium implementations, we stream in the public key to cache more intermediate results. We discuss the suitability of the signature schemes for streaming, adapt existing implementations, and compare performance.

Keywords: NISTPQC · Cortex-M3 · Signature Verification · Streaming · Post-Quantum Signatures · Memory-Constrained Devices

1 Introduction

The generally larger keys and signatures of post-quantum signature schemes have enormous impact on cryptography on constrained devices. This is especially important when the payload of the signed message is much smaller than the signature, due to additional transmission overhead required for the signature. Such short messages are for example used in the real-world use case of feature activation in the automotive domain. Feature activation is the remote activation of features that are already implemented in the soft- and hardware of the car. For example, an additional infotainment package. Usually, a short activation code is protected with a signature to prevent unauthorized activation of the feature.

In the automotive sector, it is very common to perform all cryptographic operations on a dedicated hardware security module (HSM) that resembles a Cortex-M3 processor with a clock frequency of 100 MHz and limited memory resources, e.g., [13]. Typically, the HSM is in the same package as the main processor with its own memory and is connected via an internal bus with a bus speed of about 20 Mbit/s. A fair estimate for available memory for signature verification on the HSM is under 18 kB of RAM and 10 kB of flash. However, we aim for a lower memory usage of 8 kB of RAM and flash to allow additional space for other applications and an operating system.

In this scenario signatures are verified in the very constrained environment of an HSM. It may not be able to store large public keys or keep large signatures in memory. Sometimes even the main processor does not have sufficient memory resources. Then the public key or signature must be provided to the HSM by another device in the vehicle network, like the head unit. In this case, the public key or signature must be streamed in portions over the in-vehicle network to the destination processor. A typical streaming rate over the CAN bus of an in-vehicle network is about 500 kbit/s, considering a low error transmission rate. Appendix A provides more details on the use case.

Contribution In this work, we address the challenge of performing signature verification of post-quantum signature schemes with a large public key or signature in a highly memory-constrained environment. Our approach is to stream the public key or the signature.[1] We show that this way signature verification can be done keeping only small data packets in constrained memory. When streaming the public key, the device needs to securely store a hash value of the public key to verify the authenticity of the streamed public key. During signature verification, the public key is incrementally hashed, matching the data flow of the streamed public key. We implemented and benchmarked the proposed public key and signature streaming approach for four different signature schemes (Dilithium, SPHINCS$^+$, Rainbow, and GeMSS). Although for Dilithium streaming the public key is not strictly necessary, the saved bytes allow us to keep more intermediate results in memory. This results in a speed-up.

For comparison, we also implemented the lattice-based scheme Falcon for which streaming small data packets is not necessary in our scenario as the entire public key and signature fit into RAM. The source code is published and available at `https://git.fslab.de/pqc/streaming-pq-sigs`. We demonstrate that the proposed streaming approach is very well suited for constrained devices with a maximum utilization of 8 kB RAM and 8 kB Flash.

Related Work To the best of our knowledge, this is the first work that addresses signature verification by streaming in the public key or signature. For

[1] Appendix B sketches an alternative scheme that relies on symmetric cryptography with device-specific keys. This would fit even more constrained environments, but comes at the expense of the downsides of symmetric key management.

signature schemes, streaming approaches have been investigated in [14] but the focus of that work was on signature generation (for stateless hash-based signatures). The encryption scheme Classic McEliece was studied for constrained device, solving the issue of public keys being larger than the available RAM by either streaming [29,30] or placing them in additional Flash [7,10].

2 Analyzed Post-Quantum Signature Schemes

We now briefly discuss the different signature schemes that we considered. Our exposition is focused on signature verification due to limited space. For all schemes we selected parameters that meet at least NIST security level 1. Where possible we prioritized verification speed over signature speed as we assume that signatures are created on devices that are significantly more powerful than the ones we consider for signature verification.

2.1 Hash-based Schemes

For hash-based signature schemes security solely relies on the security properties of the cryptographic hash function(s) used. Hash-based signatures can be split into stateful and stateless schemes. Stateful schemes require that a user keeps a state with the secret key. The stateful schemes LMS and XMSS are already specified as RFCs [20,15] and standardized by NIST [8]. As these schemes have sufficiently small signatures and keys, we do not consider them in this work.

SPHINCS$^+$ SPHINCS$^+$ is the last remaining stateless hash-based signature scheme in the NIST competition [3]. In the following we give a rough overview of SPHINCS$^+$ signature verification and motivate our parameter choice. For a high-level description of SPHINCS$^+$ see appendix C.

SPHINCS$^+$ signature verification consists of four components. First, the message compression, second message mapping functions, third computing hash chains, and fourth verifying authentication paths in binary hash trees. The message mapping functions take negligible time compared to the other operations and also only minimally increase space. Hence, they are ignored in our exposition. Message compression consumes an n-bit randomizer value from the signature in addition to the message which can in theory be streamed in chunks of the internal block size of the used hash function. The resulting message digest is mapped to a set of indices used later to decide the ordering of hash values in the authentication path verifications. Hash chain computation consumes one n-bit hash value from the signature and iterates the hash function a few times on the given value. The results of 67 hash chain computations are compressed using one hash function call. Hence, results have to be kept in memory until one block for the hash function is full. Finally, authentication path computation takes the n-bit result of a previous computation and consumes one authentication path node per tree level. In theory, these computations can be done one-by-one which would allow streaming each n-bit node separately.

SPHINCS$^+$ is defined as a signature framework with a magnitude of different instantiations and parameter sets. SPHINCS$^+$ defines parameters for three different hash functions: SHA-3, SHA-256, and Haraka. We chose a SHA-256 parameter set due to its performance, well understood security, and widely deployed hardware support. Moreover, SPHINCS$^+$ defines *simple* and *robust* parameters. We chose *simple* as it matches the security assumptions of the schemes that we compare to and has better performance. Lastly, the SPHINCS$^+$ specification [1] proposes $\underline{\text{f}}$ast and $\underline{\text{s}}$mall parameters, the former optimized for signing speed, the latter for signature size. However, the small parameters have better verification speed. We chose to implement `sphincs-sha256-128s-simple` and `sphincs-sha256-128f-simple` to allow for a comparison and show what is possible when reduced signing speed is not an issue. For these SPHINCS$^+$ parameters, signing speed on a general purpose CPU is about a factor 16 slower for the `s`-parameters [3]. All internal hash values in SPHINCS$^+$ have $n = 16$ bytes for the parameters we use. Public keys are $2n = 32$ bytes. Hence, they can easily be stored on the device without any compression.

2.2 Multivariate-based Schemes

Multivariate signature schemes are based on the hardens of finding solutions to systems of equations in many variables over finite fields, where the degree of the equations is at least two. The first multivariate signature scheme was designed by Matsumoto and Imai [19] and broken by Patarin [22]. Patarin with several coauthors went on to design modified schemes [23,26,17] which form the basis of modern multivariate signature schemes.

To fix notation, let the system of equations be given by m equations in n variables over a finite field $\mathbb{F}_q$. Most systems use multivariate quadratic (MQ) equations, i.e. equations of total degree two. Then the m polynomials have the form

$$f_k(x_1, x_2, \ldots, x_n) = \sum_{1 \le i \le j \le n} a_{i,j}^{(k)} x_i x_j + \sum_{1 \le i \le n} b_i^{(k)} x_i + c^{(k)} \tag{1}$$

with coefficients $a_{i,j}^{(k)}, b_i^{(k)}, c^{(k)} \in \mathbb{F}_q$.

Let M be a message and let $H : \{0,1\}^* \times \{0,1\}^r \to \mathbb{F}_q^m$ be a hash function. A signature on M is a vector $(X_1, X_2, \ldots, X_n) \in \mathbb{F}_q^n$ and a string $R \in \{0,1\}^r$ satisfying for all $1 \le k \le m$ that $f_k(X_1, X_2, \ldots, X_n) = h_k$ for $H(M,R) = (h_1, h_2, \ldots, h_m)$. The inclusion of R is necessary because not every system has a solution.

Verification is conceptually easy – simply test that all signature equations hold. Signing depends on the type of construction and what information the signer has to permit finding a solution to the system, but this is outside the scope of this paper.

General considerations for streaming MQ systems lead to short signatures but the public keys need to contain the coefficients of (1) and are thus very large, in the range of a few hundred kB. The public keys can be streamed in in blocks of rows or columns, depending on how the public key is represented. At most m elements of $\mathbb{F}_q$ are needed to hold the partial results of evaluating $f_k(X_1, X_2, \ldots, X_n), 1 \leq k \leq m$ in addition to the n elements for the signature and the m elements for the hash.

Rainbow Rainbow [9] is a finalist in round 3 of the NIST competition. Rainbow uses two layers of the Oil and Vinegar (OV) scheme [24]. For Rainbow the finite field is $\mathbb{F}_{2^4}$, so signatures require $\lceil m/2 \rceil$ bytes, leading to 66 bytes in NIST security level 1. We implement `rainbowI-classic` rather than one of the "circumzenithal" or "compressed" variants. Public keys are 158 kB for `rainbowI-classic`.

In Rainbow, the coefficients b and c are all zero. During verification, we load in columns $a_{i,j}^{(*)}$ corresponding to coefficients of each monomial $x_i x_j$, $i \leq j$. If $0 \neq x_i x_j = k \in \mathbb{F}_{16}$, we accumulate $a_{i,j}$ into a column $\mathbf{A}_k$, If $x_i = 0$, we skip all columns involving x_i. The final result is $\sum_{k \in \mathbb{F}_{16}^*} k\mathbf{A}_k$.

GeMSS GeMSS [6] is an alternate in round 3 of the NIST competition. GeMSS is based on the HFEv$-$ scheme [25]. For GeMSS the finite field is $\mathbb{F}_2$, so signatures are very small, starting at 258 bits for category I, but to achieve security the public key needs to be very large, starting at 350 kB for category I.

It bears mentioning that GeMSS is special among multivariates in that it employs the Patarin-Feistel structure to achieve very short signatures, wherein a public key is used *four* times during the verification. With pubkey f being m equations in n variables, to verify the signature of the message $\mathbf{M}$, we do:

1. write the signature as $(\mathbf{S}_4, \mathbf{X}_4, \mathbf{X}_3, \mathbf{X}_2, \mathbf{X}_1)$ where $\mathbf{S}_i$ are size m and the $\mathbf{X}_i$ are size $n - m$ (so the actual length of the signature is $4n - 3m$).
2. At stage i, which goes from 4 to 1, we set $\mathbf{S}_{i-1} = f(\mathbf{S}_i \| \mathbf{X}_i) \oplus \mathbf{D}_i$, where $\mathbf{D}_i$ is the first m bits of $(\mathrm{SHA} - 3)^i(\mathbf{M})$.
3. The signature is valid if $\mathbf{S}_0$ is the zero vector.

There are three types of GeMSS parameters. "RedGeMSS" uses very aggressive parameters; "BlueGeMSS" uses more conservative parameters. Just "GeMSS" falls in the middle, and this is what we choose to implement. The parameter set targeting NIST security level 1 is `gemss-128` and has 350 kB public keys.

2.3 Lattice-based Schemes

Lattice-based cryptographic schemes are promising post-quantum replacements for currently used public-key cryptography since they are asymptotically efficient, have provable security guarantees, and are very versatile, i.e., they offer far more functionality than plain encryption or signature schemes.

Lattice-based signature schemes are constructed using one of two techniques, either the GPV framework that is based on the hash-and-sign paradigm [11], or the Fiat-Shamir transformation [18]. The security of lattice-based signature schemes can be proven based on hard lattice problems (usually the LWE problem, the SIS problem, and variants thereof) or the NTRU assumption.

Dilithium Dilithium is a NIST round 3 finalist [2]. Signature verification for Dilithium works as follows: The public key $pk = (\rho, \mathbf{t_1})$ consists of a uniform random 256-bit seed ρ, which expands to the matrix of polynomials $\mathbf{A}$, and $\mathbf{t}_1$. For MLWE samples $\mathbf{t} = \mathbf{As} + \mathbf{e}$, $\mathbf{t}_1$ is the first output of the Power2Round procedure [2, Figure 3], and $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}_q(\mathbf{t}, d)$ is the straightforward bit-wise way to break up an element $r = r_1 \cdot 2^d + r_0$, where $r_0 = r \mod 2^d$ and $r_1 = (r - r_0)/2^d$ with $-2^d/2 \leq r_0 < 2^d/2$. Hence, the coefficients of $\mathbf{t}_0$ are the d lower order bits and the coefficients of $\mathbf{t}_1$ — the second part of the public key — are the $\lceil \log q \rceil - d$ higher order bits of the coefficients of $\mathbf{t}$. To verify a signature $(\mathbf{z}, \mathbf{h}, c)$ for a message M, one computes $\mathbf{w}' = \mathbf{Az} - c \cdot \mathbf{t_1} \cdot 2^d$, uses the hint vector $\mathbf{h}$ to recover $\mathbf{w_1'} = \text{UseHint}(\mathbf{h}, \mathbf{w}')$, and finally verifies that $c = h(h(h(\rho||t_1)||M)||\mathbf{w_1'})$. For the details, we refer to [2].

All Dilithium parameter sets use $q = 2^{23} - 2^{13} + 1$ and $d = 13$. Hence, while the coefficients of $\mathbf{t}$ need 23 bits, the coefficients of the public key $\mathbf{t}_1$ need only 10 bits. We use the smallest instance of Dilithium, which is NIST level 2 parameter set `dilithium2`. The public key size of `dilithium2` in total is 1 312 bytes and a signature needs 2 420 bytes.

Falcon Falcon, too, is a NIST round 3 finalist [28]. Falcon's signature verification works as follows: A signature for message M, consisting of the tuple (r, s), can be verified given the public key $h = gf^{-1} \pmod q$, where $f, g \in \mathbb{Z}_q[x]/(\phi)$ for a modulus q and a cyclotomic polynomial $\phi \in \mathbb{Z}[x]$ of degree n. Firstly r and M are concatenated and hashed into a polynomial c and s is decompressed using a unary code into s_2. Then, $s_1 = c - s_2 h$ is computed and it is verified that (s_1, s_2) has a small enough norm $(\leq \lfloor \beta^2 \rfloor)$. Coefficients are compressed one-by-one and hence can be decompressed individually. The embedding norm that is computed in [28, Algorithm 16, line 6] can be computed in linear time and only requires two coefficients at a time. However, the preceding polynomial multiplication requires all coefficients of one operand to be present, preventing coefficient-by-coefficient streaming for both the signature and the public key at the same time. If, however, only the signature or the public key is streamed, the polynomial multiplication could be performed. We use `falcon-512`, targeting NIST level I, which uses dimension $n = 512$ and $q = 12289 \approx 2^{14}$, hence each coefficient of the public key needs 14 bits.

3 Implementation

The following section describes the implementations of the signature schemes for the use case of feature activation described in Section 1. The signature verification is performed on a Cortex-M3. The consumption for program flash should be limited to 8 kB. The RAM usage should not exceed 8 kB. The bus speed for streaming is assumed to run at either 500 kbit/s or 20 Mbit/s.

3.1 Streaming Interface

Signed messages and public keys are streamed into the embedded Cortex-M3 device. To avoid performance overhead, our streaming implementation follows a very simple protocol. In a first step, the length of the signed message is transmitted to the embedded device. Then the embedded device initializes streaming by supplying a chunk size to the sender and additionally supplies if signed message or public key is to be streamed first. After every chunk, the embedded device can request a new chunk or return a verification result. The chunk size may be altered between chunks, but the public key and the signed message are always streamed in-order.The result is a one-bit message, signaling if the verification succeeded or failed, followed by the message in case the verification succeeded.

3.2 Public Key Verification

As the public key is being streamed in from an untrusted source, it is imperative to validate that the key is actually authentic. We assume that a hash of the public key is stored inside the HSM in some integrity-protected area.While the public key is being streamed in, we incrementally compute a hash of it that we eventually compare with the known hash. We use the same hash function as used by the studied scheme, i.e., SHA-256 for `sphincs-sha256` and `rainbowI-classic`, SHAKE-128 for `gemss-128` and SHAKE-256 for `dilithium2` and `falcon-512`. We keep the hash state in memory, occupying additional 200 bytes for SHAKE-128 and 32 bytes for SHA-256. We use the incremental SHA-256 and SHAKE implementations from pqm4 [16].

In the case of `gemss-128`, the public key is needed multiple times; once in every of the four evaluations of the public map. Note that the integrity needs to be verified each time.

3.3 Implementation Details

In the following, we describe the modifications to existing implementations of the five studied schemes needed to use them with the given platform constraints. Table 1 lists the public key, signature sizes, and the time needed for streaming them into the device at 500 kbit/s and 20 Mbit/s.

SPHINCS$^+$ Our SPHINCS$^+$ implementation is based on the round-3 reference implementation [1]. Preceding work [16] shows that computation time for SPHINCS$^+$ verification on single-core embedded devices is almost exclusively spent in the underlying hash function. We did therefore not investigate further optimization possibilities. Aligning the implementation to a streaming API is fairly straightforward as SPHINCS$^+$ signatures get processed in-order. For both `sphincs-sha256-128f-simple` and `sphincs-sha256-128s-simple` a public key is 32 bytes and hence does not require streaming.

For `sphincs-sha256-128f-simple`, a signature is 17 088 bytes. The selected SPHINCS$^+$ parameter sets use $n = 16$ byte and so a streaming chunk size of 16

Table 1. Communication overhead in bytes and milliseconds at 500 kbit/s and 20 Mbit/s. GeMSS requires to stream in the public key *nb_ite* times (4 for `gemss-128`). All other schemes require streaming in the public key and signed message once.

	streaming data			streaming time	
	$\|pk\|$	$\|sig\|$	total	500 kbit/s	20 Mbit/s
`sphincs-s`[a]	32	7 856	7 888	126.2 ms	3.2 ms
`sphincs-f`[b]	32	17 088	17 120	273.9 ms	6.9 ms
`rainbowI-classic`	161 600	66	161 666	2 586.7 ms	64.7 ms
`gemss-128`	352 188	33	1 408 785[c]	22 540.6 ms	563.5 ms
`dilithium2`	1 312	2 420	3 732	59.7 ms	1.5 ms
`falcon-512`	897	690	1 587	25.4 ms	0.6 ms

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple` [c] $4 \cdot |pk| + |sig|$

bytes is possible. However, such a small chunk is undesirable due to overhead in terms of memory and computation. The leading 16 bytes of the signature make up the randomizer value, followed by the 3 696 byte FORS signature, consisting of 33 authentication paths, and the 13 376 bytes for 22 MSS signatures. Our implementation first processes a 3 712 byte chunk containing the randomizer value and FORS signature. This is used to compute the message digest and then the FORS root, evaluating the 33 authentication paths. Then, the computed FORS root is verified using the MSS signatures. The overall 22 MSS signatures, each consisting of a WOTS+ signature and an authentication path, are processed in three chunks. Given the memory constraints, the largest available chunk size is 4 864 bytes containing 8 MSS signatures. MSS signature streaming is therefore done in two 4 864 byte chunks and one final 3 648 byte chunk. Starting from the FORS root, this data is used to successively reconstruct all the MSS tree roots from the respectively previous root: first computing 67 hash chains using the WOTS+ signature, compressing their end nodes in a single hash, and then evaluating an authentication path. The last (or "highest") MSS tree root is then compared to the root node in the public key. For this to work, the reserved chunk buffer needs to be 4 864 bytes large.

For `sphincs-sha256-128s-simple`, streaming works analogously. The signature size is 7 856 bytes and only seven - slightly larger - MSS signatures are used within the scheme. This makes it possible to stream in all 7 MSS signatures in a single 4 928 bytes chunk. Streaming therefore consists of one FORS+randomizer-value (2 928 bytes) and one MSS (4 928 bytes) chunk.

Rainbow The round-3 submission of Rainbow [9] contains an implementation targeting the Cortex-M4. As it relies only on instructions also available on the Cortex-M3, it is also functional on the Cortex-M3. However, due to the large public key (162 kB), we adapt the implementation for streaming. Rainbow signatures consist of an ℓ bit (128 for `rainbowI-classic`) salt and n (100) variables x_i in a small finite field ($\mathbb{F}_{16}$ for `rainbowI-classic`). Two $\mathbb{F}_{16}$ elements are packed into one byte in the signature and public key. We first unpack the elements of the signature and store one x_i in the lower four bits of a byte. This doubles memory usage from 50 to 100 bytes, but makes look-ups for individual elements easier. After the signature and corresponding x_i are stored in memory, the public key is streamed in. The public key consists of the Macaulay matrix $p_{i,j}^{(k)}$ representing the public map consisting of m (64) equations of the form

$$p^{(k)}(x_1, \ldots, x_n) = \sum_{i=1}^{n} \sum_{j=i}^{n} p_{i,j}^{(k)} x_i x_j$$

with the x_i, x_j corresponding to the variables from the signature. For computational efficiency the public key is represented in the column-major form. The public key's first 32 byte chunk therefore has the form $[p_{1,1}^{(1)}|p_{1,1}^{(2)}| \ldots |p_{1,1}^{(m)}]$ and the contained coefficients should be multiplied by x_1^2. Subsequent 32 byte chunks have the same form ($[p_{1,2}^{(1)}| \ldots |p_{1,2}^{(m)}]$ should be multiplied by $x_1 \cdot x_2$ and so forth.). To increase performance, Rainbow implementations delay multiplications. Before the actual multiplication step, coefficient sums are accumulated. Every incoming 32 byte chunk is added to one of 15 accumulators a_k based on the 15 possible values of $y_{i,j} = x_i \cdot x_j$ with $y_{i,j} > 0$. If $y_{i,j}$ is zero, the chunk is discarded. Once all chunks are consumed, every accumulator is multiplied by its corresponding factor $\tilde{a}_k = [k \cdot a_k^{(1)}|k \cdot a_k^{(2)}| \ldots]$ and added summed up the final result.

One can exploit that if an element x_i is zero, all monomials $x_i x_j$ will be zero and the corresponding columns of the public key will not contribute to the result. As every 16th x_i is expected to be zero, this results in a significant speed-up. As Rainbow is using $\mathbb{F}_{16}$ arithmetic, additions are `XOR`. For multiplications, we use the bitsliced implementation from the Rainbow Cortex-M4 implementation [9].

The smallest reasonable chunk size for Rainbow is a single column of the Macaulay matrix, i.e., 32 bytes. However, as larger chunk sizes result in lower overhead, we use the largest chunk size which fits in our available memory. Due to the low memory footprint of the Rainbow implementation, we can afford to use chunks of 214 columns, i.e., 6 848 bytes. As there are 5 050 ($n \cdot (n+1)/2$) columns, the last chunk is only 128 columns, i.e., 4 096 bytes.

GeMSS To the best of our knowledge, there are no GeMSS implementations available targeting microcontrollers and we, hence, write our own. We base our GeMSS implementation on the reference implementation accompanying the specification [6]. The biggest challenge is that the entirety of the 352 kB public key is needed in each of the evaluations of the public map **p**. Due to the iterative construction of the HFEv- scheme, there appears to be no better approach than streaming in the public key in each iteration, i.e., *nb_ite* (4 for `gemss-128`) times.

Each application of the public map p requires the computation of

$$p_i = \sum_{i=0}^{n+v} \sum_{j=i}^{n+v} x_i x_j a_{i,j} + a_0.$$

Each column of the Macaulay matrix needs to be multiplied by a product of two variables and then added to the accumulator. The field used by GeMSS is

$\mathbb{F}_2$ and, hence, field multiplication is logical `AND` and field addition is `XOR` which allows straightforward bitslicing of operations. Unfortunately, since the number of equations (m) is not a multiple of 8 ($m = 162$ for `gemss-128`), one cannot simply store the Macaulay matrix in column-major form since this would result in the columns not being aligned to byte boundaries. Therefore, GeMSS stores the first $8 \cdot \lfloor m/8 \rfloor$ (160) equations in a column-major form making up the first $\lfloor m/8 \rfloor \cdot (((n+v) \cdot (n+v+1))/2+1)$ (347 840) bytes of the public key with $n+v$ ($n = 174, v = 12$) being the number of variables. The last 2 equations are stored row-wise occupying the last $2 \cdot (((n+v) \cdot (n+v+1))/2+1)/8$ (4348) bytes.

We split the computation in two parts: The first $8 \cdot \lfloor m/8 \rfloor$ equations and the last ($m \mod 8$) equations. For the former, the most important optimization comes from the observation that if either of the two variables x_i or x_j is zero, the corresponding column does not impact the result. Similar to the Rainbow implementation, in the case x_i is zero, the entire inner loop and, hence, $n+v-i$ columns of the public key can be skipped. As half of the x_i are expected to be zero, this results in a vast performance gain. For the last ($m \mod 8$), we first compute the monomials $x_i x_j$ and store them in a vector, then we add this vector to each row of the public key. Lastly, we compute the parity of each row. The smallest reasonable chunk size for the first part of the computation is one column of the public key (20 bytes), while it is one row (2174 bytes) for the second part. However, we use 4 560 byte-chunks (285 columns) to achieve lowest overhead with 8 kB RAM.

Dilithium. Our Dilithium implementation is based on previous work targeting the Cortex-M3 and Cortex-M4 [12]. However, this work predates the round 3 Dilithium submission [2] which introduced some algorithm tweaks and parameter changes. Most notably for the performance of `dilithium2` verification, the matrix dimension of $\mathbf{A}$ changed from $(k, \ell) = (4, 3)$ to $(4, 4)$. Therefore, we adapt the existing Cortex-M3 implementation to the new parameters.

For `dilithium2`, the implementation of [12] requires 9 kB of stack in addition to the 2.4 kB signature and 1.3 kB public key in memory. We apply a couple of tricks to fit it within 8 kB: We compute one polynomial of $\mathbf{w'}$ at a time, which allows us to stream in the public key $\mathbf{t_1}$. Usually, one computes $\mathbf{w'} = \mathbf{A}\mathbf{z} - c \cdot \mathbf{t_1} \cdot 2^d$ as $\texttt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \texttt{NTT}(\mathbf{z}) - \texttt{NTT}(c) \cdot \texttt{NTT}(\mathbf{t_1} \cdot 2^d))$. Hence, it is desirable for performance to keep $\texttt{NTT}(\mathbf{z})$ and $\texttt{NTT}(c)$ in memory. However, that already occupies 5 kB. We instead keep the compressed forms of $\mathbf{z}$ and $\mathbf{c}$ in memory, occupying only $\ell \cdot 576 = 2304$ and 32 bytes, respectively, and recompute the `NTT` operations.

Previous implementations of Dilithium use 3 temporary polynomials to compute $\texttt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \texttt{NTT}(\mathbf{z}) - \texttt{NTT}(c) \cdot \texttt{NTT}(\mathbf{t_1} \cdot 2^d))$, one for the accumulator and two temporary ones for the inputs. We instead compute $\texttt{NTT}^{-1}(-\texttt{NTT}(c) \cdot \texttt{NTT}(\mathbf{t_1} \cdot 2^d) + \hat{\mathbf{A}} \cdot \texttt{NTT}(\mathbf{z}))$, which can be computed in 2 polynomials by sampling $\hat{\mathbf{A}}$ coefficient-wise, as was also proposed for Kyber [5].

The total memory consumption comprises the 2 420-byte signature, 2 polynomials of 1 024 bytes each, 3 Keccak states of 200 bytes each, and about 600 bytes of other buffers, i.e., approximately 5 670 bytes in total. To improve speed, one can cache as much of $\texttt{NTT}(\mathbf{z})$ and $\texttt{NTT}(\mathbf{c})$ as possible. We cache $\texttt{NTT}(\mathbf{c})$ and 3 polynomials of $\texttt{NTT}(\mathbf{z})$ while still remaining below 8 kB of stack.

Falcon We used a Cortex-M4 optimized implementation which is also part of Falcon's round-3 submission [27,28]. It is compatible with Cortex-M3 processors, but relies on emulated floating point arithmetic. This leads to data-dependent runtimes, which is unproblematic for verification, but may be an issue when considering signing as well. On the Cortex-M3, the implementation submitted to NIST uses around 500 bytes of stack space, public keys of circa 900 bytes, signatures of around 800 bytes, and a 4 kB scratch buffer. The overall memory footprint is about 6.5 kB. Hence, streaming in the data in small packets is not necessary. Our implementation copies the whole public key and signature to RAM before running the unmodified Falcon verification algorithm.

4 Results

We chose an ARM Cortex-M3 board with 128 kB RAM and 1 MB Flash, an STM32 Nucleo-F207ZG, as the platform for the implementation of our case study. This board meets most of the specifications of an environment with limited resources of a typical automotive HSM embedded in MCUs. The only mismatch is the non-volatile memory (NVM). A typical limited HSM has much less NVM.

We clock the Cortex-M3 at 30 MHz (rather than the maximum frequency of 120 MHz) to have no Flash wait states. In a practical deployment in an HSM one would use fast ROM instead of Flash and, hence, our cycle counts are close to what we would expect in an automotive HSM.

We base our benchmarking setup on the pqm3[2] framework and adapt it to support our streaming API. For counting clock cycles, we use the SysTick counter. We stream in the signed message and public key using USART, but disregard the cycles spent waiting for serial communication. We stream in the signed message and public key using USART using a baud rate of 57 600 bps, which is much slower than what we would expect in a practical HSM. We use `arm-none-eabi-gcc` version 10.2.0 with `-O3`. We use a random 33-byte message which resembles the short messages needed for feature activation.

Table 2 presents the speed results for our implementations. The studied signature schemes rely on either SHA-256 (`rainbowI-classic`, `sphincs-sha256`) or SHA-3/SHAKE (`dilithium2`, `falcon-512`, and `gemss-128`). In a typical HSM-enabled device SHA-256 would be available in hardware and SHA-3/SHAKE will also be available in the future. However, on the Nucleo-F207ZG no hardware accelerators are available. Hence, we resort to software implementations instead. For SHA-256 we use the optimized C implementation from SUPERCOP.[3] For SHA-3/SHAKE, we rely on the ARMv7-M implementation from the XKCP.[4]

[2] https://github.com/mupq/pqm3
[3] https://bench.cr.yp.to/supercop.html
[4] https://github.com/XKCP/XKCP

Table 2. Cycle count for signature verification for a 33-byte message. Average over 1 000 signature verifications. Hashing cycles needed for verification of the streamed in public key (hashing and comparing to embedded hash) are reported separately. We also report the verification time on a practical HSM running at 100 MHz and also the total time including the streaming at 20 Mbit/s.

	w/o pk vrf.	w/ pk verification			w/ streaming
		pk vrf.	total	time[e]	20 Mbit/s
`sphincs-s`[a]	8 741k	0	8 741k	87.4 ms	90.6 ms
`sphincs-f`[b]	26 186k	0	26 186k	261.9 ms	268.7 ms
`rainbowI-classic`	333k	6 850k[d]	7 182k	71.8 ms	136.5 ms
`gemss-128`	1 619k	109 938k[c]	111 557k	1 115.6 ms	1 679.1 ms
`dilithium2`	1 990k	133k[c]	2 123k	21.2 ms	21.8 ms
`falcon-512`	581k	91k[c]	672k	6.7 ms	8.2 ms

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple` [c] SHA-3/SHAKE
[d] SHA-256 [e] At 100 MHz (no wait states)

While GeMSS and Rainbow only compute a (randomized) hash of the message, SPHINCS$^+$, Dilithium, and Falcon use hashing as a core building block of the verification. Consequently, the amount of hashing in multivariate cryptography is minimal (2% for `rainbowI-classic`, 4% for `gemss-128`), while it makes up large parts for lattice-based (65% for `dilithium2`, 36% for `falcon-512`) and hash-based signatures (90% for `sphincs-sha256-128s-simple` and 88% for `sphincs-sha256-128f-simple`). Clearly lattice-based and hash-based schemes would benefit more from hardware accelerated hashing.

Additionally, we need to verify the authenticity of the streamed in public key. We report the time needed for public key verification separately. For hash-based signatures this operation comes virtually for free as the public key itself can be stored in the device, so that no hashing is required. For multivariate cryptography, the public key verification becomes the most dominant operation due to the large public keys and fast arithmetic. This is particularly pronounced for GeMSS as the public key is the largest and needs to be verified 4 times.

Table 3 presents the memory requirements of our implementations.

Acknowledgments

Part of this work was done during the Lorentz Center Workshop "Post-Quantum Cryptography for Embedded Systems", Oct. 2020. This work was carried out while the third and fifth author were visiting Academia Sinica. This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE) and by the German Federal Ministry of Education and Research (BMBF) under the project QuantumRISC. Part of Marc Stöttinger's contribution to this work was done while he was employed at Continental AG. This work was supported by Taiwan Ministry of Science and Technology Grant 109-2221-E-001-009-MY3, Sinica Investigator Award AS-IA-109-M01, Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP). This

Table 3. Memory and code-size requirements in bytes for our implementations. Memory includes stack needed for computations, global variables stored in the .bss section and the buffer required for streaming. Code-size excludes platform and framework code as well as code for `SHA-256` and `SHA-3`.

	memory				code
	total	buffer	.bss	stack	.text
`sphincs-s`[a]	6 904	4 928	780	1 196	2 724
`sphincs-f`[b]	7 536	4 864	780	1 892	2 586
`rainbowI-classic`	8 168	6 848	724	596	2 194
`gemss-128`	8 176	4 560	496	3 120	4 740
`dilithium2`	8 048	40	6 352	1 656	7 940
`falcon-512`	6 552	897	5 255	400	5 784

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple`

work has been supported by the Netherlands Organisation for Scientific Research (NWO) under grant 628.001.028 (FASOR).

References

1. Aumasson, J.P., Bernstein, D.J., Beullens, W., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Westerbaan, B.: SPHINCS+. Submission to the NIST Post-Quantum Cryptography Standardization Project (2020), available at https://sphincs.org/
2. Bai, S., Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-DILITHIUM. Submission to the NIST Post-Quantum Cryptography Standardization Project (2020), available at https://pq-crystals.org/dilithium/
3. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS$^+$ Signature Framework. In: ACM CCS. ACM (2019). https://doi.org/10.1145/3319535.3363229
4. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: Practical Stateless Hash-Based Signatures. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT, Lecture Notes in Computer Science, vol. 9056, pp. 368–397. Springer (2015), https://eprint.iacr.org/2014/795
5. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-efficient high-speed implementation of Kyber on Cortex-M4. In: Buchmann, J., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology – Africacrypt 2019. Lecture Notes in Computer Science, vol. 11627, pp. 209–228. Springer (2019), https://eprint.iacr.org/2019/489
6. Casanova, A., Faugère, J.C., Macario-Rat, G., Patarin, J., Perret, L., Ryckeghem, J.: GeMSS. Submission to the NIST Post-Quantum Cryptography Standardization Project (2020), available at https://www-polsys.lip6.fr/Links/NIST/GeMSS.html
7. Chen, M.S., Chou, T.: Classic McEliece on the ARM Cortex-M4. Cryptology ePrint Archive, Report 2021/492 (2021), https://eprint.iacr.org/2021/492

8. Cooper, D., Apon, D., Dang, Q., Davidson, M., Dworkin, M., Miller, C.: NIST Special Publication 800-208: Recommendation for stateful hash-based signature schemes (2020). https://doi.org/10.6028/NIST.SP.800-208
9. Ding, J., Chen, M.S., Kannwischer, M., Patarin, J., Petzoldt, A., Schmidt, D., Yang, B.Y.: Rainbow. Submission to the NIST Post-Quantum Cryptography Standardization Project (2020), available at https://www.pqcrainbow.org/
10. Eisenbarth, T., Güneysu, T., Heyse, S., Paar, C.: Microeliece: Mceliece for embedded devices. In: Clavier, C., Gaj, K. (eds.) CHES. pp. 49–64. Springer (2009). https://doi.org/10.1007/978-3-642-04138-9_4
11. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: ACM STOC. p. 197–206. STOC '08, ACM (2008). https://doi.org/10.1145/1374376.1374407
12. Greconici, D.O.C., Kannwischer, M.J., Sprenkels, D.: Compact Dilithium implementations on Cortex-M3 and Cortex-M4. IACR TCHES **2021**(1), 1–24 (2020), https://eprint.iacr.org/2020/1278
13. Henniger, O., Ruddle, A., Seudié, H., Weyl, B., Wolf, M., Wollinger, T.: Securing vehicular on-board IT systems: The EVITA project. In: 25th Joint VDI/VW Automotive Security Conference (Oct 2009), https://evita-project.org/Publications/HRSW09.pdf
14. Hülsing, A., Rijneveld, J., Schwabe, P.: ARMed SPHINCS. In: PKC. p. 446–470. Springer (2016)
15. Hülsing, A., Butin, D., Gazdag, S.L., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. RFC 8391 (May 2018). https://doi.org/10.17487/RFC8391
16. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Workshop Record of the Second NIST PQC Standardization Conference (2019), https://eprint.iacr.org/2019/844
17. Kipnis, A., Patarin, J., Goubin, L.: Unbalanced oil and vinegar signature schemes. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 1592, pp. 206–222. Springer (1999). https://doi.org/10.1007/3-540-48910-X_15
18. Lyubashevsky, V.: Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In: ASIACRYPT. pp. 598–616. Springer (2009). https://doi.org/10.1007/978-3-642-10366-7_35
19. Matsumoto, T., Imai, H.: Public quadratic polynominal-tuples for efficient signature-verification and message-encryption. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 330, pp. 419–453. Springer (1988). https://doi.org/10.1007/3-540-45961-8_39
20. McGrew, D., Curcio, M., Fluhrer, S.: Leighton-Micali Hash-Based Signatures. RFC 8554 (Apr 2019). https://doi.org/10.17487/RFC8554
21. Merkle, R.: A certified digital signature. In: CRYPTO. Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer (1990). https://doi.org/10.1007/0-387-34805-0_21
22. Patarin, J.: Cryptanalysis of the Matsumoto and Imai public key scheme of Eurocrypt'88. In: CRYPTO. Lecture Notes in Computer Science, vol. 963, pp. 248–261. Springer (1995). https://doi.org/10.1007/3-540-44750-4_20
23. Patarin, J.: Hidden fields equations (HFE) and isomorphisms of polynomials (IP): two new families of asymmetric algorithms. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 1070, pp. 33–48. Springer (1996). https://doi.org/10.1007/3-540-68339-9_4

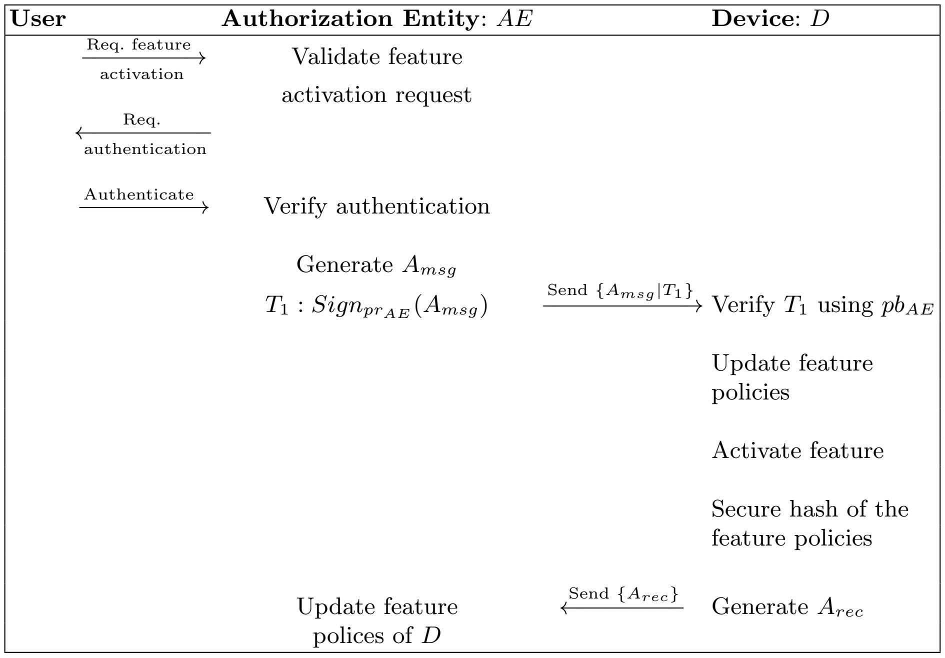
24. Patarin, J.: The oil and vinegar signature scheme. In: Dagstuhl Workshop on Cryptography (Sep 1997)
25. Patarin, J., Courtois, N.T., Goubin, L.: Quartz, 128-bit long digital signatures. In: CT-RSA. Lecture Notes in Computer Science, vol. 2020, pp. 282–297. Springer (2001). https://doi.org/10.1007/3-540-45353-9_21
26. Patarin, J., Goubin, L., Courtois, N.T.: C^*_{-+} and HM: variations around two schemes of T. Matsumoto and H. Imai. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 1514, pp. 35–49. Springer (1998). https://doi.org/10.1007/3-540-49649-1_4
27. Pornin, T.: New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Report 2019/893 (2019), https://eprint.iacr.org/2019/893
28. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Submission to the NIST Post-Quantum Cryptography Standardization Project (2020), available at https://falcon-sign.info/
29. Roth, J., Karatsiolis, E.G., Krämer, J.: Classic McEliece implementation with low memory footprint. In: CARDIS. Lecture Notes in Computer Science, vol. 12609, pp. 34–49. Springer (2020). https://doi.org/10.1007/978-3-030-68487-7_3
30. Strenzke, F.: How to implement the public key operations in code-based cryptography on memory-constrained devices. Cryptology ePrint Archive, Report 2010/465 (2010), https://eprint.iacr.org/2010/465

A Feature Activation

Feature Activation is intended to activate additional functionality on an embedded device that is already deployed and active in the running environment. It differs from a software update because all software and required hardware for the feature's functionality is already included in the device, but not activated.

The feature is activated by an authentic message from an authorized instance. The activation of the feature is device specific, therefore the activation messages must not be portable to other devices. The protocol 1 describes the actual feature activation process between an embedded device on which the feature is to be activated and an authorized instance, e.g., a back-end system. To authenticate the feature activation request, a signature is part of the message sent from the authorization instance to the embedded device. Nowadays, this signature is implemented, for example, by an ECC signature, which is not a post-quantum algorithm. In the scenario shown, the overall protocol does not take into account any resource constraints on the device, so that, for example, the ECC signature and the public key are stored entirely on the device.

The protocol 1 can be roughly paraphrased as follows: The user, e.g. the car owner, creates a request to activate a desired feature for a specific vehicle (identified by a vehicle identification number - VIN). This can be done through an online platform. The authorization instance, which can be represented by a back-end, validates the feature request for the feature policies it stores for the requested vehicle, and requests and verifies the user's authentication. Upon successful authorization, the authorization instance generates a device-specific feature activation request A_{msg} for the device that is part of the vehicle and


User
Authorization Entity: AE
Device: D
$\xrightarrow[\text{activation}]{\text{Req. feature}}$
Validate feature activation request
$\xleftarrow[\text{authentication}]{\text{Req.}}$
$\xrightarrow{\text{Authenticate}}$
Verify authentication
Generate A_{msg}
$T_1 : Sign_{pr_{AE}}(A_{msg})$
$\xrightarrow{\text{Send } \{A_{msg}|T_1\}}$
Verify T_1 using pb_{AE}
Update feature policies
Activate feature
Secure hash of the feature policies
Update feature polices of D
$\xleftarrow{\text{Send } \{A_{rec}\}}$
Generate A_{rec}

Protocol 1: Protocol for feature activation

implements the requested feature. Furthermore, the authorization instance generates a signature T_1 for the message A_{msg} using its private key pr_{AE}. When the device successfully verifies the signature T_1, it updates its feature policies, activates the requested feature, and stores the feature policy hash. Finally, the embedded device confirms the feature activation status in a message A_{rec} to the authorization instance. The authorization instance itself also updates and stores the feature policy for the specific device.

B Alternative Implementation

For embedded applications it is sometimes attractive to use symmetric cryptography in place of public-key cryptography. Not only is symmetric cryptography a lot faster, it also benefits from already-present hardware acceleration and key sizes are significantly smaller. Of course, the secret keys in symmetric devices are extremely sensitive. We need that a secret key extracted from a particular deployed device does not compromise the entire scheme. This implies the need to provision each device with its own individualised key. However, when deploying hundreds or thousands of devices this means we have a potentially significant key management problem. Fortunately, the many-to-one architecture in this automotive application implies we only need a single key between each device and the back-end. Furthermore, each deployed device has public identifiers, like the vehicle VIN or a serial number. This allows us to only let the manufacturer store a single key for all deployed devices.

We use these properties to construct an efficient key distribution scheme. Let each device have a unique identifing number n. This could for example be the concatenation of the vehicle VIN and the device's serial number. We let the manufacturer generate a main secret key K_m. Then, we provision at time of manufacturing each device with the following key K_d, such that

$$K_d = \mathrm{KDF}(K_m, n).$$

Here, KDF is an appropriate key-derivation function.

Then, whenever the device needs to use their key K_d in communication with the manufacturer, they send over their identifier n along with the message. For example, if they need to send an authenticated message m, they might send $\{n, m, \mathrm{MAC}_{K_d}(n, m)\}$. The manufacturer can then easily compute K_d based on n and the main secret K_m, and verify the message. As the device does not have access to K_m, they can only have produced this MAC if they were provisioned with K_d at time of manufacture.

Of course, this entire scheme falls down when K_m is compromised. As such, special care needs to be taken to protect it. Although the private keys used in public-key cryptography also need to be protected, we can use revocation mechanisms to recover from a compromise. This is not possible with symmetric cryptography.

C Hash-Based Signatures

Hash-based signature schemes are signature schemes for which security solely relies on the security properties of the cryptographic hash function(s) used. In contrast to other proposals for digital signature schemes it does not require an additional complexity theoretic hardness assumption. Given that the security of cryptographic hash functions is well understood and even more, we know that generic attacks using quantum computers cannot significantly harm the security of cryptographic hash functions, hash-based signatures present a conservative choice for post-quantum secure digital signatures. The description in this section is simplified, and we refer to the official specification [1] for a detailed exposition.

One-Time Signature Schemes (OTS) Hash-based signatures built on the concept of a one-time signature scheme (OTS). This is a signature scheme where a key pair may only be used to sign one message. If two messages are signed with the same secret key, the scheme becomes insecure. Such OTS can be constructed from cryptographic hash functions. The very generic concept is that the secret key consists of random values while the public key contains their hash values. A signature consists of a subset of the values in the secret key, selected by the message. A signature is verified by hashing the values in the signature and comparing the resulting hash values to the respective values in the public key.

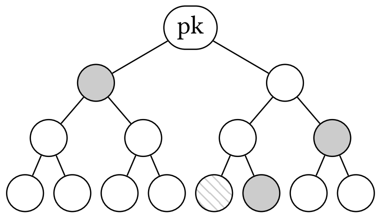


Fig. 1. The authentication path of the fifth leaf (Source [3])

The OTS commonly used today is the Winternitz OTS (WOTS) or variations thereof which generalizes the above concept to hash chains. WOTS has the important property that a signature is verified by computing a candidate public key by hashing the values in the signature several times (depending on the message) and comparing the result to the public key.

Merkle Signature Schemes (MSS) Given a OTS, a many-time signature scheme can be constructed following the concept of Merkle Signature Schemes (MSS) [21]. For these, a number (a power of 2) of OTS key pairs is generated and their public keys are authenticated using a binary hash tree, called a Merkle tree. The hashes of the public keys form the leaves of the tree. Inner nodes are the hash of the concatenation of their two child nodes. The root node becomes the MSS public key. Assuming WOTS is used as OTS, a signature consists of the leaf index, a WOTS signature and the so-called authentication path (cf. figure 1). The authentication path contains the sibling nodes on the path from the used leaf to the root. Verification uses the WOTS signature (and the message) to compute a candidate public key and from that the corresponding leaf. This leaf is then used together with the authentication path to compute a root node: Starting with the leaf, the current buffer is concatenated with the next authentication path node and hashed to obtain the next buffer value. The order of concatenation is determined by the leaf index in the MSS signature. The final buffer value is then compared to the root node in the public key.

In general, this leads a so-called stateful scheme as a signer has to remember which OTS key pairs she already used. This concept is the general idea underlying the schemes described in recent RFCs [20,15] LMS and XMSS.

SPHINCS$^+$ The limitation of having to keep a state as signer can be overcome in practice using the SPHINCS construction [4] (previous theoretically efficient proposals by Goldreich go back to the last century but were only of theoretical interest). SPHINCS$^+$ [3] essentially instantiates the SPHINCS construction. The first idea in SPHINCS uses a few-time signature scheme (FTS) - a signature scheme where a key pair can be used to sign a small number of messages without keeping a state before the scheme gets insecure. SPHINCS$^+$ uses a huge number of FTS key-pairs (in the order of 2^{64} depending on the parameters). For every new message, a random FTS key is picked to sign. By making the number of

FTS keys large enough, the probability that one key gets used to sign more than a few messages can be made vanishingly small. The public keys of all these FTS key pairs are authenticated using a certification tree of MSS key pairs called the hypertree. The hyper tree is essentially a PKI. To the top MSS key works as a root CA and the bottom layer MSS keys certify FTS public keys. The whole structure is deterministically generated using pseudorandom generators. That way, it is not necessary to store which OTS keys where used for an MSS key because the message that a specific OTS key will be used to sign is predetermined.

The FTS in SPHINCS$^+$ is FORS. A FORS secret key consists of several sets of random values the values in each set are authenticated via a Merkle tree. These trees have the hashes of the secret values as leaves. The public key is the hash of the concatenation of all root nodes of these Merkle trees. A signature consists of one secret key value from each set (determined by the message) and the respective authentication path. Verification works by computing the leaves from the signature values and afterwards computing candidate root nodes as for MSS. This can be done per tree. Afterwards, the roots are used to compute a candidate public key.

A SPHINCS$^+$ signature consists of a randomizer R that is hashed with the message to obtain the message digest, a FORS signature, and the MSS signatures on the path from the FORS keypair to the top tree. Verification computes a message digest using the message and R. The message digest is split into the index of the FORS signature and the indices of the Secret key values in the FORS signature. With this, the FORS signature is used to compute a candidate public key. This candidate FORS public key is used as message to compute a candidate MSS root node with the first MSS signature which is used as message for the next signature, and so on. The final MSS root node is compared to the root node in the public key.