Implementing SPHINCS+ on Embedded Systems: Ensuring Performance in Resource-Constrained Environments

Maya Abi Ghanem[1,2], Matthieu Courrier[1], Brice Colombier[2], and Lilian Bossuet[2]

[1] *Aumovio France, Toulouse, France*
{*maya.abi.ghanem; matthieu.courrier*}*@aumovio.com*

[2] *Université Jean Monnet Saint-Etienne, CNRS, Institut d'Optique Graduate School, Laboratoire Hubert Curien UMR 5516, F42023, Saint-Etienne, France*
{*maya.abi.ghanem; b.colombier; lilian.bossuet*}*@univ-st-etienne.fr*

ABSTRACT

Quantum computing poses a significant threat to widely used public-key cryptographic schemes such as RSA and ECDSA. Post-quantum cryptographic (PQC) algorithms like SPHINCS+, recently standardized in Federal Information Processing Standards (FIPS) 205, offer a quantum-resistant alternative but are resource-intensive and challenging to deploy on embedded systems with strict real-time constraints. In this work, we investigate the feasibility of accelerating SPHINCS+ on Traveo II embedded platforms dedicated to automotive applications, specifically the 1M (CYT2B7) variant, featuring an Arm Cortex-M4F with Cortex-M0+ cryptographic cores and integrated SHA-2 accelerators. By offloading key hash operations and leveraging multicore execution across the application and Hardware Security Module (HSM) cores, we achieve measurable performance gains. Our results show that this hybrid approach can bring SPHINCS+ execution times within limits acceptable for automotive-grade embedded systems. However, to meet stricter real-time requirements, dedicated hardware support such as FPGA-based accelerators or integrated PQC co-processors will be necessary. These findings contribute to bridging the gap between post-quantum cryptography and practical deployment in constrained embedded environments.

Keywords - Post-Quantum Cryptography, SPHINCS+, Embedded Systems, Hardware Acceleration, ARM Cortex-M, Real-Time Constraints, Automotive Security

1. INTRODUCTION

With the growing connectivity in embedded systems, particularly in the automotive industry, security has become a crucial concern. In addition to their local network of up to 150 Electronic Control Units (ECUs) [1], modern vehicles are now connected to external networks such as other vehicles and remote servers (cloud systems) to enable communication and data storage or advanced computations. This increased connectivity expands the potential attack surface, making robust security essential to ensure system operability and, by extension, user safety.

Cryptography plays a major role in securing embedded systems by ensuring data integrity, authenticity and confidentiality. As a result, hardware accelerators for widely used cryptographic standards are integrated into embedded systems and microcontrollers to meet the industry's stringent demands for low latency, high computational power, and minimal storage requirements. These cryptographic standards were designed to remain secure for decades.

However, they now face an emerging threat: the potential development of quantum computers. Though still in the early stages, quantum computing poses a significant risk to classical cryptography [2, 3], promoting the National Institute of Standards and Technology (NIST) to initiate a search for new post-quantum cryptographic standards. Unlike other candidates that have been standardized, the hash-based SPHINCS+ [4] does not require hybridization (i.e., combining classical and post-quantum algorithms [5]) for additional security due to its well-understood security properties and quantum resistance [6].

High-performance chips can handle the increased computational and storage demands of these new algorithms. However, the majority of automotive ECUs are resource constrained, and thus cannot support them while still meeting the performance and energy requirements of the automotive domain. The hardware accelerators needed to support these algorithms are often unavailable on automotive ECUs and, in many cases, not mature enough to be integrated. Nevertheless, most microcontrollers used in automotive applications are equipped with hardware hash accelerators. Thus SPHINCS+ could potentially be accelerated on existing automotive chips, without the need for a technical overhaul.

Contribution In this work, we implement SPHINCS+ with partial hardware acceleration on Infineon Traveo II T2G microcontrollers. We show that, for all the use-cases of the signature scheme (key generation, signature generation, and verification), hash computations can be offloaded to the dedicated Secure Hash Algorithm (SHA)-2 hardware accelerator, while distributing the logic across the main application core and the Hardware Security Module (HSM) core (Cortex-M0+). We evaluate this approach on the CYT2B7 platforms (Traveo II 1M) [7]. Compared to a purely software implementation, the proposed method yields a measurable improvement in execution time (51.6%), bringing SPHINCS+ within practical bounds for constrained embedded systems. While this work is tailored to Traveo II (TVII) devices, the approach is generalizable to other embedded platforms with cryptographic accelerators and heterogenous core architecture.

Outline The rest of the paper is organized as follows. Section 2 exposes the context of security, and more precisely cryptography in the embedded systems industry and reviews norms, and explores related work. A detailed presentation of SPHINCS+ and its constructions are presented in Section 3, as well as our contribution. Sections 4 and 5 detail the adopted strategies, as well as experimental setup and preliminary results, before ending with a conclusion in Section 6, accompanied with discussions.

2. Background and Context

2.1. General Automotive Context

Cybersecurity standards such as ISO 21434 [8] were established to provide threat modeling frameworks and guide secure system development. Cryptography has become a foundational defense in automotive systems, ensuring confidentiality, integrity, and authenticity across applications such as secure in-vehicle communication, Over-the-Air (OTA) updates, keyless entry, and Vehicle-To-Everything (V2X) messaging. These applications rely on a diverse set of cryptographic functions, ranging from hashing to digital signature and encryption.

Quantum computing threatens current cryptographic schemes like Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC), prompting industries to adopt quantum-resistant alternatives to ensure long-term security. This transition is reinforced by regulatory mandates. The UNECE WP.29 requires that, as of January 2025, automotive manufacturers obtain Cyber Security Management System (CSMS) certification for each vehicle platform, demonstrating compliance with up-to-date cryptographic practises [9].

In automotive systems, asymmetric cryptography is primarily used for secure boot, secure firmware update, and certificate-based authentication in V2X protocols. These use cases rely on digital signatures, making them natural candidates for post-quantum secure schemes such as SPHINCS+.

2.2. Related Work

Several studies have investigated SPHINCS+ in resource-constrained environments. Kannwischer et al. [10] provided foundational benchmarks for SPHINCS+ on low-power embedded devices, identifying key performance bottlenecks and memory usage challenges. Their work emphasizes the computational cost of SPHINCS+ and the challenges it poses for deployment on constrained devices. They concluded that hashing operations constitute the majority of computational cost in SPHINCS+, accounting for up to 98% of the overall execution time.

Karl et al. [11] examined the effects of different hash primitives and hardware acceleration strategies, showing how specific choices can reduce computation time but introduce non-negligible communication overhead between the processor and external hash units.

Niederhagen et al. [12] proposed a streaming implementation for Trusted Platform Modules (TPMs), optimizing SPHINCS+ through memory-efficient hashing and staged computation to reduce latency without requiring hardware support.

Finally, Kim et al. [13] introduced a parallelized SPHINCS+ implementation leveraging Graphics Processing Unit (GPU) thread-level parallelism, achieving significant speedups. However, their approach remains unsuitable for embedded automotive contexts, where GPUs are rare, and energy and cost efficiency are paramount.

A comparable effort is the QuantumSAR project by IAV [14, 15], which offers an AUTomotive Open System ARchitecture (AUTOSAR)-compatible PQC driver. However, Classic AUTOSAR lacks a dedicated cryptographic Microcontroller Abstraction Layer (MCAL), so deterministic execution may be affected by CPU sharing when in typical automotive use cases such as secure boot or flashing, the CPU is fully available, making some of QuantumSAR's runtime-management overhead unnecessary. The public implementation also shows limited adaptation to memory-constrained platforms or hardware acceleration, which is essential for hash-heavy schemes like SPHINCS+. Furthermore, its benchmarks on high-end devices (Aurix TC399) do not reflect performance on lower-end ECUs. Thus, although QuantumSAR demonstrates an industrial path toward PQC integration, it is not yet well suited for highly constrained or time-critical embedded environments.

In contrast, our work focuses on a more realistic deployment in automotive microcontrollers that already include hash engines and secure cores, filling a gap in the literature by aligning performance optimization with practical automotive requirements and regulatory demands.

3. SPHINCS+

SPHINCS+ is a stateless, hash-based digital signature scheme. It is an improved version of the original SPHINCS signing scheme, designed to address its limitations in signature size and computational efficiency. Hash-based signature schemes are unique in that their security relies solely on the properties of the underlying hash functions. Since hash functions (such as SHA-2 or Secure Hash Algorithm Keccak (SHAKE)) are widely believed to remain secure against quantum computers (unlike RSA or ECC which are vulnerable to Shor 's algorithm), hash-based signatures provide a strong foundation for post-quantum cryptosystems. Grover's algorithm only offers a quadratic speedup against hash functions [16], which is manageable by doubling the security level of the function. While lattice-based schemes (e.g. CRYSTALS-Dilithium or Falcon) are popular candidates for post-quantum cryptography due to their efficiency, they rely on relatively newer mathematical assumptions such as the hardness of Learning With Errors (LWE) or Shortest Vector Problem (SVP). A significant distinction of SPHINCS+ is that it is not based on lattice-based cryptography. It is solely built on well-studied and minimal assumption of hash functions security, offering a conservative and arguably trustworthy

basis. This is specially appealing for high-assurance or long-term security applications, especially where crypto-agility is limited. As noted in Hülsing et al. (2019) [17], "SPHINCS+ provides post-quantum security from minimal assumptions", making it an important hedge in the broader landscape of post-quantum cryptographic options. By offering multiple parameter sets, SPHINCS+ allows a trade-off between signature size and computational speed. This scheme is defined as a signature framework, a cryptographic construction that integrates multiple hash-based components [4]. Specifically, SPHINCS+ combines stateless Merkle tree structures with two core primitives, as shown in Figure 1: Forest of Random Subsets (FORS) for few-time signatures and Winternitz One Time Signature Plus (WOTS+) for one-time signatures. As illustrated in Figure 1, the FORS components consists of k independent Merkle trees of height a, whose roots are combined into a single FORS root. This root is then authenticated using an eXtended Merkle Signature Scheme (XMSS) hypertree, where XMSS denotes a Merkle-tree-based signature scheme built from WOTS+ leaves, composed of d stacked XMSS layers, each of height h', yielding a total hypertree height $h = h' \times d$.
This layered composition enables SPHINCS+ to achieve post-quantum security while maintaining a stateless signing process, addressing the traditional limitations of stateful hash-based signatures such as those based solely on Merkle trees.

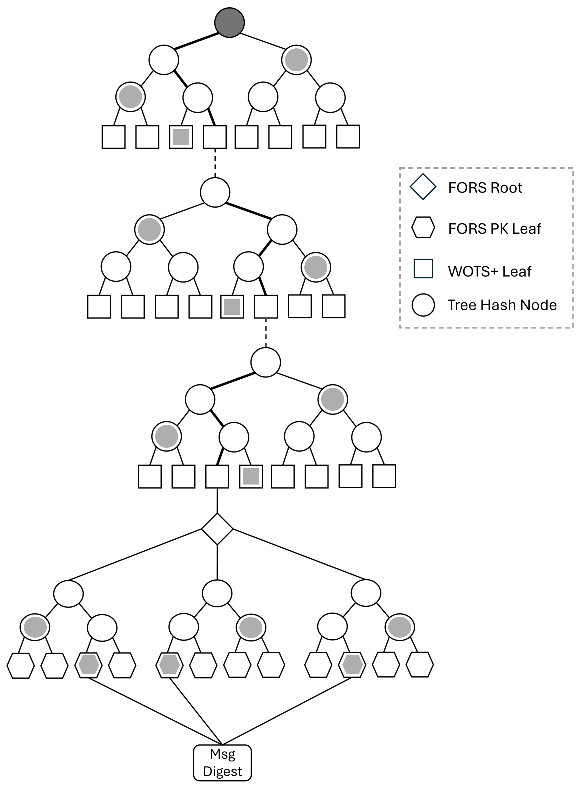

Figure 1. SPHINCS+ Hypertree for $d = 3$, $h' = 3$, $a = 2$ and $k = 3$, the upmost top node being the scheme's public key.

Merkle Tree A Merkle tree is a perfect binary tree constructed using a cryptographic hash function over a set of leaf nodes, each corresponding to a public key derived from a private signing key. It is parameterized by its height h, which determines the number of leaf nodes (2^h), and by the security parameter n, the output size in bytes of the underlying hash function, which also defines the size of each node in the tree.

The internal nodes of the tree ($2^h - 1$ nodes) are computed by iteratively hashing pairs of child nodes up to the root, which serves as the public key of the entire structure and is shared with the verifier. Signing a message involves selecting a unique private key from the set of secret keys, computing its corresponding public key (leaf), and providing an authentication path. The authentication path consists of the sibling nodes along the path from the selected leaf to the root. It enables the verifier to recompute the Merkle root (i.e., the public key) independently and confirm that the signature corresponds to a valid key within the tree.

Merkle-tree-based signatures are inherently stateful: reusing a signing key (i.e., leaf node) compromises security by enabling potential key recovery. Therefore, secure implementations must ensure that each key is used only once, tracking and updating state across signatures.

The number of hash computations required to construct a Merkle tree is given by equation (1).

$$N_{\text{Hash}}^{\text{Merkle}} = 2^{h'} - 1 \qquad (1)$$

This expression highlights the exponential dependence on the tree height h.

WOTS+ and FORS SPHINCS+ relies on two foundational hash-based signature primitives: WOTS+ and FORS, each serving a distinct role within the overall construction. WOTS+ is a one-time signature scheme that encodes a message into a sequence of hash chains, where each chain corresponds to a digit in a base-w representation. To safeguard against forgery through partial chain manipulation, WOTS+ incorporates a checksum derived from the message representation, ensuring the integrity of the full message during verification.

In the WOTS+ signature scheme, the signature is generated by applying a hash function multiple times to the secret key components, generated with a Pseudorandom Function (PRF) function in SPHINCS+'s case. Each secret key element x_i is hashed (with the hash function H) m_i times, where m_i is derived from the base-w representation of the message to be signed, therefore producing the signature component $\sigma_i = H^{m_i}(x_i)$. This process ensures that the signature is linked to both the message and the secret key, while allowing the verifier to compute the public key via its components $y_i = H^{w-1}(x_i)$ by computing the remainder of the chain, i.e. ensuring that $H^{(w-1)-m_i}(\sigma_i)$ is equal to y_i as illustrated in Figure 2, which also shows how a WOTS+ Merkle tree leaf is computed. For example, in a WOTS+ instance with $w = 4$, a message block (or message component) $m_0 = 1$ results in a $\sigma_i = H^1(x_0)$, as shown in Figure 2. Consequently, the mechanism balances efficiency and security by using a hash chain to derive verifiable signatures without exposing the full secret key. However, reusing the same secret key across multiple signatures compromises this security guarantee and can enable forgery, and must therefore be strictly avoided.

FORS complements WOTS+ by efficiently signing short messages, typically digests, using a subset of secret keys determined directly by the message. These keys, along with

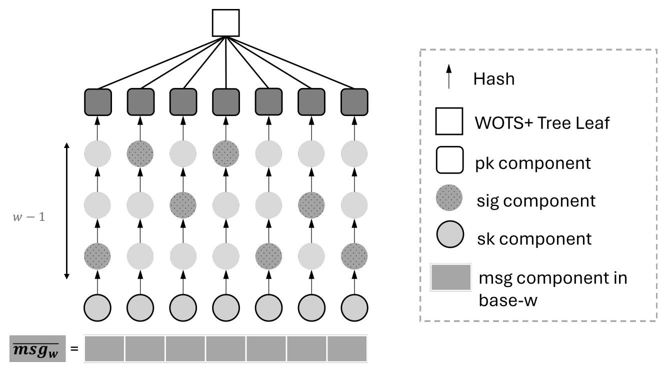

Figure 2. WOTS+ Leaf Node Generation Process in SPHINCS+, with $w = 4$ and $len = 7$.

their corresponding authentication paths, are revealed to the verifier, who reconstructs the derived public key. The resulting FORS output is then itself signed using WOTS+, forming a hierarchical, layered structure. Together, WOTS+ and FORS enable SPHINCS+ to maintain post-quantum security, support stateless signing, and achieve efficient verification without relying on number-theoretic assumptions.

To generate the FORS secret keys, the `prf_addr` function is used, producing pseudo-random values derived from the addresses in the hypertree. The equation (2) expression accounts for the total number of hash evaluations required for the entire FORS procedure, including both secret-key generation and the computation of all internal nodes in the FORS forest.

$$N_{\text{Hash}}^{\text{FORS}} = 1 + \sum_{i=1}^{k} (2 \cdot 2^a) + 2^{a-1} \tag{2}$$

As for WOTS+, the number of hash computations is significantly higher. As shown in Figure 2, each WOTS+ leaf is derived from len secret components, each generated with the `prf_addr` function.

Every component then undergoes the Winternitz chain iteration to produce both the signature element (at a position determined by the message) and the final public key component at the end of the chain, as formalised in equation (3). All len components are finally combined using a single `thash` operation to form the corresponding WOTS+ Merkle tree leaf.

$$N_{\text{Hash}}^{\text{WOTS+}} = \sum_{i=1}^{D} \sum\nolimits_{j=1}^{2^{h'}} [len + len \cdot (w-1) + 1] \tag{3}$$

Tweakable Hash Functions Tweakable hash functions are designed to enhance collision resilience. Their primary goal is to ensure that even if a collision is found in the underlying hash function, the overall security of the cryptographic scheme remains intact. In SPHINCS+, the tweakable hash function takes three inputs: a public parameter P (specifically, the public seed from the public key, padded with zeroes to complete the SHA-2 block), a tweak T (an address value representing the context within the hypertree), and the message input M. This can be denoted by the equation (4).

$$Thash(P, T, M) = Th_{P,T}(M) = H(P||T||M) \tag{4}$$

The tweakable hash function is instantiated using SHA-2. The function processes the public seed as the first block, padded to the full block size, followed by blocks containing the tweak (typically an address in the hypertree structure) and the message input. While the SPHINCS+ specification supports multiple underlying hash functions, we selected SHA-2 due to its widespread support in embedded environments and the availability of an efficient optimization. This optimization involves precomputing and exporting the internal SHA-2 state after hashing the public seed once, then reinjecting this state during subsequent evaluations. By factoring out the static first block, this technique reduces redundant computations and improves performance, a particularly valuable enhancement for resource-constrained platforms such as automotive microcontrollers.

Key Pair A SPHINCS+ key pair consists of a public key (PK) and a private key (SK). The public key contains two n-byte values :

- The public seed, used as input to the tweakable hash function (providing domain separation and determinism).
- The root of the top-level Merkle tree in the hypertree structure.

The private key includes :

- The secret seed, used to deterministically derive the private components of the WOTS+ and FORS keys using a PRF.
- The secret PRF seed, used to compute the pre-signature message randomisation value R.
- The public seed and top-level root (duplicated from the public key), which are included to allow signature generation without needing the public key as input.

This design allows for stateless signing, eliminating the need to maintain per-signature state, while ensuring forward secrecy and enabling deterministic generation of keys and signatures.

Signature A SPHINCS+ signature has three parts:

1. Message randomization value, derived using the PRF seed and the message.
2. FORS signature, which signs a digest of the randomized message. The FORS scheme is used for its compact and parallelizable structure.
3. Hypertree authentication path, which includes:
    - A WOTS+ signature of the FORS public key.
    - A sequence of WOTS+ signatures authenticating each level of the Merkle hypertree, each accompanied by the respective authentication paths.

To verify a signature, the verifier reconstructs each level of the hypertree using the WOTS+ public keys and their authentication paths, ascending level by level until the root of the hypertree is computed. If the resulting root matches the root included in the public key, the signature is valid. This layered tree structure (hypertree) allows SPHINCS+ to achieve high security without relying on stateful operations, making it robust and suitable for long-term secure applications.

	n	h	d	h'	a	k	lg_w	len	**security category**	**pk bytes**	**sig bytes**
SPHINCS+-SHA2-128s	16	63	7	9	12	14	4	30	1	32	7 856
SPHINCS+-SHA2-192s	24	63	7	9	14	17	4	39	3	48	16 224
SPHINCS+-SHA2-256s	32	64	8	8	14	22	4	47	5	64	29 792

Table 1. SPHINCS+ Parameters and Sizes with SHA-2

It is important to note that earlier versions of SPHINCS+ included a “simple” parameter set, which did not include input masking in its hash functions. This omission increased its vulnerability to certain multi-target attacks. As a result, only the robust parameter set, which does employ input masking, has been standardized by the NIST. The robust variant offers a stronger security margin and is the only recommended configuration for deployments. This article focuses on this configuration set.

Since robust `thash` generates one mask per input block, the total number of hash evaluations required for mask generation is a function of the input length and contributes non-negligibly to the overall hashing cost. The mask generation cost is given by the equation (5).

$$N_{\text{Hash}}^{\text{mgf1}} = \left\lceil \frac{inblocks \cdot n}{\text{sha256\_output\_bytes}} \right\rceil \tag{5}$$

Equation (6) formalises the number of additional hashes required by WOTS+ for hashing the leaf concatenations in each subtree, while Equation (7) represents the additional hashes required for computing the internal nodes of the Merkle tree.

$$N_{\text{mgf1\_hash}}^{\text{WOTS+}} = \sum_{i=1}^{2^{h'}} (len \cdot (w-1) + \lceil \frac{len \cdot n}{sha2\_output\_bytes} \rceil) \tag{6}$$

$$N_{\text{mgf1\_hash}}^{\text{Merkle}} = \sum_{i=1}^{D} ((2^{h'}-1) \cdot \left\lceil \frac{2n}{\text{sha256\_output\_bytes}} \right\rceil) \tag{7}$$

The same logic applies to FORS public keys, their internal nodes, and their root computations, which incur similar mask-generation overheads.

4. Implementation Strategies

Implementing SPHINCS+ on embedded platforms presents a range of design challenges, primarily due to its intensive memory requirements and the large number of hash function calls involved, often numbering in the millions per signature. The reference SPHINCS+ code was used as a basis for this work, with minor adaptations for the embedded platform. Due to confidentiality restrictions, the modified implementation cannot be publicly released. However, all algorithmic modifications and design details are fully described in this article, enabling reproducibility without disclosing the source code.

4.1. Platform Characteristics

In this work, we target the Traveo II 1M microcontroller, a widely used automotive-grade platform, and explore software and hardware co-design techniques to optimize performance and memory usage within its constrained environment.

The Infineon TVII 1M microcontroller (CYT2B7) was selected for this work due to its widespread adoption in the automotive industry and its integrated hardware cryptographic accelerator supporting SHA-2. The platform features a 160 MHz 32-bit Arm Cortex-M4F CPU for application tasks and a 32-bit Arm Cortex-M0+ CPU hosting the HSM firmware and the cryptographic accelerator, as illustrated in Figure 3. Since the accelerator is accessible only through the CM0+ core, cryptographic operations initiated by the application core must be routed via remote procedure calls, introducing latency and synchronization constraints.

In the TVII, the Inter-Processor Communication (IPC) is implemented through a mailbox mechanism, which provides a lightweight register-based message passing interface. The HSM core then serializes these requests and forwards them to the crypto accelerator, and the same procedure applies to the responses, as shown in Figure 3.

The available RAM on the platform is 128 kB, which imposes constraints on stack usage and dynamic memory allocation.

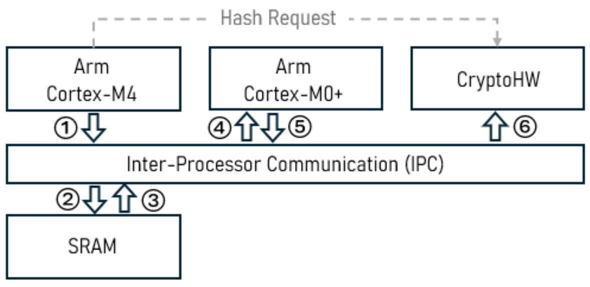

Figure 3. CPU subsystem of the TVII-1M CYT2B7, highlighting the serialization path of a hardware hash request. The response follows the same path in reverse. The HSM firmware is hosted on the Cortex-M0+.

4.2. Parameter Set Selection

We employ the robust variant of SPHINCS+. We opt for the 128-bit security level with the ”slow” variant. Although the ”fast” variant reduces key generation and signing time significantly, it nearly doubles the signature size and increases verification time, which we consider to be an impractical trade-off for memory-constrained embedded systems in automotive applications.

The hash function used throughout our implementation is SHA-256, which aligns well with the platform's 32-bit architecture and benefits from hardware acceleration. We also exclude Haraka hash function from consideration: while it offers performance advantages on x86 platforms with AES-NI support, it lacks Federal Information Processing Standards (FIPS) standardization and does not benefit from the instruction sets available on our target hardware.

4.3. Hash Function Analysis and Optimization

4.3.1. Quantifying Hashing Cost in SPHINCS+

The number of hash operations in the SPHINCS+ signature framework is directly determined by the selected parameter set. During keypair generation, only the top-level Merkle tree is computed and the FORS construction is not invoked. As a result, the total number of hash evaluations is substantially smaller than in signature generation, since only a single layer of the SPHINCS+ hypertree is processed, and this is formalized by the equation (8).

$$\begin{aligned} N_{hash}^{Merkle}(D=1) + N_{mgf1\_hash}^{Merkle}(D=1) \\ + N_{hash}^{WOTS+}(D=1) + N_{mgf1\_hash}^{WOTS+}(D=1) \end{aligned} \quad (8)$$

During signature generation, the message is first hashed and preprocessed to derive a per-signature randomization value. This ensures that signatures depend on the specific message and that signing the same message twice yields different outputs. Next, the FORS forest is constructed, and its root is obtained. This FORS root is then signed using WOTS+, whose leaf nodes are inserted into a Merkle tree to produce a corresponding Merkle root. That root is itself signed by another WOTS+ instance in the layer above. This process of generating a WOTS+ signature and authenticating it through a Merkle tree repeats for each layer of the hypertree until the top-level tree is reached. The root of this final tree constitutes the second half of the overall PK of the SPHINCS+ scheme. The equation (9) formalizes the number of hashes required in a signature generation :

$$\begin{aligned} N_{hash}^{Merkle}(D=d) + N_{mgf1\_hash}^{Merkle}(D=d) \\ + N_{hash}^{WOTS+}(D=d) + N_{mgf1\_hash}^{WOTS+}(D=d) \\ + N_{hash}^{FORS} + N_{mgf1_hash}^{FORS} \\ + N_{hash}^{preprocess} + N_{mgf1\_hash}^{preprocess} \end{aligned} \quad (9)$$

The computational cost of signature verification is significantly lower than that of key generation and signature creation. However, unlike these two operations, the number of hash evaluations required during verification cannot be expressed as a single fixed value, as it depends on the specific signature being verified. This variability arises from the remaining number of WOTS+ chain steps that must be computed to reach each WOTS+ public-key element during verification. Nevertheless, the verification cost can be bounded between well-defined lower and upper limits, allowing the overall complexity of the operation to be rigorously characterized.

Merkle tree authentication takes as many hash operations as the height of said Merkle tree. This is applied also for the FORS trees. In total, for the authentication of Merkle trees (including the FORS trees) that would amount to :

$$N_{\text{hash}}^{\text{Merkle}} = d \cdot 2^{h'} + k \cdot 2^{a} \quad (10)$$

When verifying a WOTS+ signature, only a single leaf, and therefore only one WOTS+ hash chain, in each tree layer needs to be partially recomputed. The amount of work varies because the number of hash steps required depends on the message-derived value for that chain.

As explained in Section 3, the hash chain length is dependant on the message component value. While it is uncommon that the signature for a specific component coincide with the secret key (if $m_i = 0$) or the public key (if $m_i = w - 1$, the maximal value of a message component), it is not impossible. This would make the number of hashes for the WOTS+ verification would be between situated anywhere between 0 (if the signature were to coincide with the SK component) or $w - 1$ (if it were to be the PK component). So in total, the number of hash for the whole hypertree is given by equation (11).

$$0 \leq N_{hash}^{WOTS+\_verify} \leq 2 \times \sum_{i=1}^{d} \sum_{j=1}^{len} (w-1) \quad (11)$$

This is augmented by the fixed hash costs associated with preprocessing, Merkle tree verification, and the FORS scheme.

4.3.2. Hash Implementation Optimization

As discussed in Section 3, SPHINCS+ relies heavily on a `thash` function, which forms the backbone of FORS, WOTS+, and Merkle tree constructions. Each call hashes a message combined with a context-dependent tweak and a public seed. We leverage a known optimization for SHA-256 in software, where the padded initial state derived from the public seed is precomputed and reused across calls, effectively factoring out the seed processing and reducing computational cycles. To illustrate the optimization, Figure 4 provides a schematic view of the SHA-256 Merkle–Damgård construction. In the diagram, the internal chaining state obtained after processing the padded public seed is highlighted in black. This state is the one exported by our implementation and reused across subsequent hashing operations, allowing the tweak and message blocks to be injected directly into the compression function without reprocessing the seed.

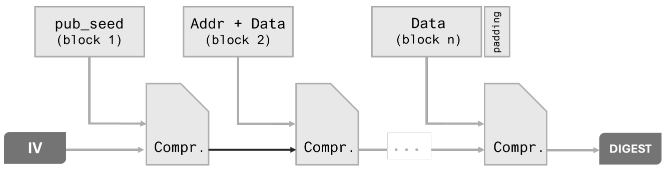

Figure 4. SHA-2 Merkle–Damgård Construction Showing the Preloaded State Derived from the Public Seed.

In our implementation, we extend this approach to the hardware crypto-accelerator by preloading the SHA-256 state corresponding to the padded seed. This hybrid strategy preserves the functional security of tweakable hashing while providing a significant performance improvement.

As formalized in Section 4.3.1, the total number of hash function calls in SPHINCS+ is fixed for a given message. The optimization presented here does not reduce the number of hashes, but instead reduces the number of internal blocks that must be processed in each hash by precomputing and reusing the SHA-256 state derived from the public seed. This distinction is crucial: the security of the scheme remains unchanged, while computational efficiency is significantly improved.

Algorithm 1 SPX_Sha256_Seed_State(PubSeedPtr, SeededStatePtr)

```
1: $seed\_block[SHA256\_BLOCK\_SIZE] \leftarrow \{0x00\}$
2: for $i = 0$ to $n$ do
3:     $seed\_block[i] = PubSeedPtr[i]$
4: end for
5: Sha256_Init() ▷ Init Sha256 with IV
6: Sha256_Update(seed_block , SHA256_BLOCK_SIZE ) ▷ Sha256 Hashblock with block containing seed + zeroes padding
7: Export_State(SeededStatePtr) ▷ State is extracted from the cryptoaccelerator internal memory
8: return $SeededStatePtr$
```

Algorithm 1 shows the procedure for exporting the precomputed SHA-256 state. This allows subsequent hash calls to reuse the padded seed state without recomputing it, as if simply resuming a hash operation. The `SPX_Sha256_Seed_State` function takes as input the public seed and writes a serialized hash state that is stored or transmitted for later reuse.

Algorithm 2 SPX_sha256_Seeded(DigestPtr, SeededStatePtr , InPtr, Inlen)

```
1: $processed\_bytes \leftarrow 64U$
2: Import_State(SeededStatePtr) ▷ State is imported to the cryptoaccelerator internal buffer
3: Sha256_Update(InPtr , Inlen )
4: Sha256_Finish(DigestPtr)
5: return DigestPtr
```

In Algorithm 2, the precomputed state is injected into the hash chain in place of the standard initialization vector. The procedure then completes the hash computation by incorporating the context-dependent tweak and the message data, producing the final `thash` output.

It is important to note that exporting and importing states does not reveal any secret key material; the procedure preserves the functional security of the hash chain while improving computational efficiency.

4.3.3. Applicability to Other Platforms

SHA-256 accelerators are usually block-level and partially software-managed.

Cryptographic accelerators are typically designed to implement block-level components of cryptographic functions rather than the entire function in hardware, while higher-level computations, synchronization, and finite-state machine control are managed in software. For example, AES accelerators often implement individual rounds in hardware, while the higher-level algorithm is executed in software. Similarly, RSA hardware modules implement modular exponentiation, leaving other control to the software component. The same principle applies to SHA-256: hardware accelerators generally provide block-level operations, while the rest of the hashing function is managed in software.

As a result, the export and import of intermediate SHA-256 states is often feasible on such accelerators, provided the chip architecture allows access to the internal state (as is not typically the case when the vendor restricts access). This means that the proposed optimization can be applied to a wide range of platforms that provide block-level cryptographic acceleration. Most importantly, the optimization is not restricted to ARM architectures; the ARM Cortex-M platform was chosen for benchmarking convenience, and the approach is broadly applicable to any compatible accelerator.

4.4. Memory Management

Dynamic memory allocation is particularly challenging in real-time embedded systems due to fragmentation, non-deterministic execution time, and the unpredictability inherent in traditional heap allocators. The reference SPHINCS+ implementation uses variable-length arrays (VLAs), which internally depend on heap allocation (e.g., via `malloc`). A naive strategy is to simply reserve static buffers sized for the worst-case temporary memory required across all of SPHINCS+ operations. Because this worst-case bound corresponds to the maximum depth of nested hash computations, it is significantly larger than what most executions require, leading to substantial and prohibitive memory overhead on embedded platforms.

To address this, we introduce a lightweight, embedded-friendly dynamic memory allocator that provides deterministic, stack-like (Last In First Out (LIFO)) allocation semantics, the pseudocode is detailed in Algorithm 3.

The allocator consists of a statically allocated byte array and a pointer that acts as a moving stack top pointer. Each allocation advances the pointer, while de-allocation simply restores it to a previous position, proportional to the freed memory size. This approach supports nested allocations, required by a deep stack, which are common operations such as tree hashing and signature composition. The total size of the memory pool is computed at compilation time from SPHINCS+ parameters bounds, ensuring that all dynamic allocations are safe and that no overflows can occur in memory-constrained environments. Although this mechanism is technically dynamic allocation, it operates over a dedicated, fixed-size region with fully deterministic behavior, making it significantly more predictable and analyzable than a general-purpose `malloc` or `alloca`, which rely respectively on the heap and the system stack. Unlike them, our allocator provides bounds checking and a fixed memory footprint. This integration also allows for the reference SPHINCS+ code to be integrated with minimal modifications in regards of memory footprint and logic.

Algorithm 3 VLA_Allocator: Stack-Based Dynamic Allocation

```
1:  global $HEAP$[0 .. N-1] ▷ statically allocated memory pool
2:  global $sp \leftarrow 0$ ▷ stack pointer
3:  function VLA_ALLOC(size)
4:      if $sp + size > N$ then
5:          error "out of memory"
6:      end if
7:      ptr ← &$HEAP$[sp]
8:      sp ← sp + size
9:      return ptr
10: end function
11: function VLA_MARK
12:     return sp ▷ save current stack pointer for rollback
13: end function
14: function VLA_RELEASE(mark)
15:     sp ← mark ▷ free all allocations since mark
16: end function
```

4.5. WOTS+ Leaf Generation and Performance Considerations

In the SPHINCS+ hypertree, each sublayer root is authenticated using a WOTS+ signature. Consequently, all leaf nodes in the Merkle trees of the hypertree are derived from WOTS+ public keys. As illustrated in Figure 2, each WOTS+ leaf is constructed through a multi-step process: first, secret key components sk_is are generated using a PRF seeded with the private seed. Each sk_i is then iteratively hashed through a Winternitz hash chain of length $w = 16$, with the output at a given position r_i yielding the signature component $H^{r_i}(sk_i)$, where $r_i < w$ is determined by the corresponding message $\log_2 w$-bit block value. The public key component for index i is $H^w(sk_i)$, and the concatenation of all such components is hashed once more to produce the WOTS+ leaf node.

The total number of hash function invocations per WOTS+ leaf is significant, neighbouring the 565,000 hashes for key generation: generating each sk_i requires one PRF call, each of the len components undergoes w applications of the hash function via `thash`, and the resulting public key is hashed again to yield the leaf. This process is repeated for all 2^h leaves in each Merkle tree layer. Furthermore, `thash` includes a masking function for domain separation and side-channel resistance, doubling the number of hash calls per invocation.

Empirical measurements show that WOTS+ operations, particularly leaf generation and signing, dominate the computational cost of SPHINCS+. In our implementation, they account for up to 98% of the total key generation time.

5. Experimental Results and Discussion

5.1. Experimental Results

As mentioned in the section 4, the TVII platform has an integrated SHA2 accelerator. However, the offloading mechanism incurs significant latency due to inter-core communication, which can outweigh the benefits of hardware acceleration. This effect is particularly visible in SPHINCS+, where the large number of hash evaluations magnifies communication overhead.

Previous work has investigated software-level optimizations for SPHINCS+, especially for SHA-2, where intermediate computations can be factorized to eliminate redundant work.

Version	KeyGen (s)	Sign (s)	Verify (ms)
OPT $_{\text{CM4}}$	28.948	216.209	223.351
NO OPT$_{\text{CM4}}$	40.817	305.591	310.541
ACC-HSM-SS$_{\text{CM4}}$	31.807	238.806	244.102
OPT $_{\text{CM0+}}$	131.46	986.752	972.124
NO OPT$_{\text{CM0+}}$	300.186	2259	2.219k
ACC-HSM-SS$_{\text{CM0+}}$	144.273	1083	1.086k
DUALCORE-HSM-SS	21,429	166,875	190
ACC-HSM-OPT$_{\text{CM4}}$	28.706	215.626	235.936
ACC-HSM-OPT$_{\text{CM0+}}$	13.933	107.528	110.503
DUALCORE-HSM-OPT	14.245	113.062	128.842

Table 2. Execution time of different SPHINCS+ configurations across the three use cases. HSM-SS denotes the vendor-intended single-shot call of the hash accelerator through the HSM interface, where each hash operation is executed independently. HSM-OPT denotes our modified HSM driver enabling hash optimizations, such as state reuse and operation chaining.

When such optimizations are applied, executing the hash function directly on the main Cortex-M4F core consistently outperforms delegating the computation to the hardware accelerator. This is further exacerbated by SPHINCS+'s reliance on tweakable hash functions, designed to make each hash dependent on its context and tree instance [4]. This structure restricts the applicability of traditional acceleration strategies and requires disabling many internal optimizations when the hardware accelerator is used, ultimately increasing execution time.

To evaluate the performance of SPHINCS+ on the Infineon TVII 1M platform, we implemented and benchmarked multiple versions of the algorithm across different cores, optimizations, and acceleration strategies. The pure software versions, both optimized and non-optimized, were executed on the Cortex-M4 and Cortex-M0+ cores, resulting in four baseline configurations. Hardware acceleration was applied to both cores, yielding two additional configurations, and a dual-core setup—where the SPHINCS+ framework runs on the CM4 while `thash` and WOTS+ operations, executed on the CM0+, provide an additional performance point. Finally, HSM-mediated acceleration was evaluated on the CM0+, CM4, and in a dual-core configuration, for three further variants. In total, ten distinct implementations were benchmarked, providing a comprehensive view of the impact of software optimization, hardware acceleration, core placement, and HSM mediation on execution time, as summarized in Table 2.

All experiments were conducted on the Traveo II 1M microcontroller in a bare-metal configuration, without an operating system or AUTOSAR services. The implementation was compiled using the Green Hills MULTI toolchain and the Green Hills C/C++ compiler (GHS), with the program linked to run entirely from on-chip Random Access Memory (RAM) using the platform's default linker script. No compiler optimization flags were enabled (i.e., no `-O*` options). Execution times were measured using an external Lauterbach TRACE32 system with cycle-accurate hardware tracing and negligible measurement overhead. All measurements were performed with interrupts disabled, and the reported values correspond to the median of multiple runs.

5.2. Discussion

The significant execution overhead observed in some configurations motivated an exploration of potential mitigation strategies. One approach is to port the entire SPHINCS+ algorithm to the Cortex-M0+ to eliminate inter-processor communication. However, this would likely result in sub-optimal performance, as the M0+ operates at approximately half the frequency of the main Cortex-M4F application CPU. Consequently, the performance gains achieved by reducing IPC overhead would be offset by slower hash and arithmetic computations.

A more effective strategy involves selective partitioning of the algorithm. For example, bypassing system calls to the crypto drivers and enabling direct execution of hash-intensive operations from the application core could reduce overhead. Additionally, offloading only the most overhead-intensive components, such as the WOTS+ scheme, to the HSM provides substantial performance benefits while maintaining cryptographic integrity. This dualcore arrangement, with the framework on the high-frequency CM4 and thash/WOTS+ operations executed on the CM0+ near the crypto accelerator, achieves a balance between throughput and secure execution in the trusted domain.

Evaluating HSM-mediated acceleration revealed substantial performance improvements, particularly when executed from the Cortex-M0+. In this configuration, the thash and WOTS+ operations leverage the crypto accelerator within the secure domain, resulting in execution times up to 89.4% faster than the optimized software baseline on CM0+, and 51.6% than that on the CM4. The dualcore configuration where the SPHINCS+ framework remains on the higher frequency CM4 while hash intensive operations execute on the CM0+ further amplifies these gains by reducing inter-processor communication overhead. In contrast, performing the same HSM-accelerated operations from the CM4 alone provides limited improvement, as the lower proximity to the crypto accelerator and the need to transmit multiple hash requests incur additional latency. These results highlight that core placement and hardware-assisted execution are critical factors in achieving efficient post-quantum cryptography on constrained embedded platforms.

Although the CM0+ operates at a lower frequency and is architecturally less suited for complex arithmetic or memory-intensive workloads, these factors did not hinder performance in our accelerated configurations. On the contrary, when combined with the crypto accelerator and positioned within the HSM domain, the CM0+ delivered the best execution times, achieving up to a 89.4% improvement compared to the optimized software version on the CM4. This confirms that the dominant limitation in the system is not the computational capability of the CM0+, but rather the inter-processor communication and serialization overhead introduced when cryptographic operations are invoked from the CM4. Despite these constraints, the resulting performance remains suitable for practical embedded use cases, particularly signature verification in secure boot and secure firmware update workflows, where verification latency is the primary concern. The presented architecture therefore provides a viable and efficient near-term solution for deploying post-quantum signatures on platforms lacking dedicated PQC accelerators.

This work highlights that the dominant bottlenecks in SPHINCS+ arise from its hash-heavy components, in particular the WOTS+ scheme, which represents the majority of the computational cost in both signing and verification. By exposing these limitations on a commercial embedded microcontroller, our results provide a realistic view of the challenges involved in deploying post-quantum signatures on constrained systems. The proposed dualcore and hardware-assisted configurations should be viewed as temporary mitigation strategies, a practical “bandage” that alleviates latency without addressing the fundamental cost of WOTS+. Future work will therefore focus on developing dedicated hardware support, including FPGA prototypes and architectural extensions for tightly integrated hash engines, with the long-term goal of enabling microcontroller vendors to provide native, low-overhead primitives suitable for SPHINCS+ and other hash-based PQC schemes.

6. Conclusion

This work evaluated the feasibility of deploying the SPHINCS+ post-quantum signature scheme on embedded automotive platforms equipped with a SHA-256 hardware accelerator and asymmetric multicore architecture. We analyzed both software-only and accelerator-assisted implementations and examined the impact of core placement, inter-core communication, and SPHINCS+ state optimized hashing on overall performance. Our results show that, when the accelerator is used without SPHINCS+-specific hash-state optimization, the dualcore configuration provides the best performance because heavy hashing tasks are offloaded to the core hosting the hardware security module, reducing the communication burden on the fast application core. In contrast, once state-optimized hashing is supported directly on the accelerator, the dominant source of overhead becomes inter-core communication itself. In this setting, executing SPHINCS+ primarily on a single core tightly coupled to the accelerator yields the most efficient solution, outperforming the dual-core approach. We acknowledge that this configuration is not fully optimal for real-time systems, and further performance improvements would likely require a dedicated SPHINCS+ accelerator, particularly given that the largest overhead in SPHINCS+ arises from WOTS+ operations. Nevertheless, the proposed single-core, accelerator-centric setup provides a practical interim solution. Notably, in typical automotive scenarios, the most frequent SPHINCS+ operation is signature verification, and our performance measurements indicate that this use case can be supported adequately on embedded platforms with the current hardware. This conclusion applies across all SPHINCS+ operations, including key generation, signature creation, and signature verification. In summary, careful software placement and close integration with the accelerator are key to achieving practical post-quantum performance in embedded automotive systems. Furthermore, future automotive-grade hardware security modules will benefit from integrating SPHINCS+ optimized hashing paths natively, enabling crypto-agile and real-time adequate post-quantum deployments.

Acknowledgment

This work received funding from the France 2030 program, managed by the French National 732 Research Agency under grant agreement No. ANR-22-PETQ-0008 PQ-TLS.

References

[1] Miroslaw Staron. *Automotive Software Architectures - An Introduction, Second Edition*. Springer, 2021.

[2] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[3] Vasileios Mavroeidis, Kamer Vishi, Mateusz D. Zych, and Audun Jøsang. The impact of quantum computing on present cryptography. *CoRR*, abs/1804.00200, 2018.

[4] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ signature framework. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2129–2146. ACM, 2019.

[5] Nina Bindel, Jonas Brendel, Marc Fischlin, Renato Gonçalves, and Douglas Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. Cryptology eprint archive, report 2018/903, International Association for Cryptologic Research, 2018.

[6] ANSSI. Avis de l'ANSSI sur la migration vers la cryptographie post-quantique. Technical report, 2024.

[7] Infineon Technologies AG. 32-bit arm® cortex®-m4f microcontroller traveo™ ii t2g family. Technical report, 2021. Datasheet.

[8] ISO/SAE 21434. Road vehicles functional safety. Technical Report 8309, 2021.

[9] UNECE. Uniform provisions concerning the approval of vehicles with regards to cyber security and cyber security management system. Number 155, 2025.

[10] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM cortex-m4. *IACR Cryptol. ePrint Arch.*, page 844, 2019.

[11] Patrick Karl, Jonas Schupp, and Georg Sigl. The impact of hash primitives and communication overhead for hardware-accelerated SPHINCS+. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, volume 14595 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2024.

[12] R. Niederhagen, J. Roth, and J. Wälde. Streaming SPHINCS+ for embedded devices using the example of TPMs. In *Progress in Cryptology - AFRICACRYPT 2022*, pages 269–291, Cham, 2022. Springer Nature Switzerland.

[13] D. Kim, H. Choi, and S. C. Seo. Parallel implementation of SPHINCS+ with GPUs. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(6):2810–2823, June 2024.

[14] IAV GmbH (Ingenieurgesellschaft Auto und Verkehr). QuantumSAR: AUTOSAR Driver Implementation. 2023. GitHub repository, accessed May 2025.

[15] P. Jungklass, M. Siebert, and D. Oka. Post-quantum cryptography on embedded ecus. In *JSAE Technical Papers*, 2024.

[16] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing (STOC '96)*, pages 212–219, New York, NY, USA, July 1996. Association for Computing Machinery.

[17] A. Hülsing. Wots+ – shorter signatures for hash-based signature schemes. Technical Report 2017/965, IACR ePrint Archive, 2017.