Politecnico di Torino

Politecnico di Torino

Master's Degree in Cybersecurity Engineering

A.y. 2024/2025

Graduation Session October 2025

Evaluating the Secure Boot Implementations using Quantum-Proof Signature Algorithms in Real-Time Embedded Systems

Supervisors:

Prof. Stefano Di Carlo

Prof. Alessandro Savino

Ing. Franco Oberti

Candidate:

Francesco Corvaglia

Ai miei genitori

Abstract

The advent of quantum computing poses a fundamental threat to widely deployed public-key cryptographic algorithms, including those currently securing firmware integrity and authenticity in embedded systems. As a result, post-quantum digital signature schemes must be investigated and integrated into critical security mechanisms such as secure boot. This thesis focuses on evaluating the performance and practicality of the Leighton–Micali Signature (LMS) scheme, a hash-based post-quantum digital signature algorithm standardized by NIST and the IETF, within the secure boot context of real-time embedded systems.

An initial comparative analysis of standardized post-quantum signature algorithms identified LMS as a favorable candidate due to its minimal and fixed key sizes, strong security assumptions based solely on hash functions, and relative implementation simplicity compared to lattice-based or stateless hash-based alternatives. LMS was then implemented in software on a Xilinx PYNQ-Z2 platform, based on the reference code, and adapted for single-core, single-threaded execution to ensure reproducibility and precise performance measurement.

To address the hashing bottleneck inherent in LMS signing and verification, a SHA-256 hardware accelerator was ported to the platform and integrated through an AXI-Lite interface. A custom Linux device driver was developed and subsequently optimized using memory-mapped I/O to reduce per-call latency. The resulting software–hardware co-design was benchmarked extensively across multiple LMS parameter sets, using execution time measurements, cycle-level profiling, and statistical analysis over 30 independent rounds per configuration.

The results show that hashing dominates LMS computation, with signature generation and key generation times scaling predictably with tree height and Winternitz parameters. Driver optimization significantly reduced I/O overhead, and cycle-level testbenches enabled accurate characterization of intrinsic accelerator latency. A break-even frequency model was constructed to estimate the operating conditions under which hardware acceleration outperforms software hashing. Additionally, LMS performance was compared against a pre-quantum ECDSA–SHA256 baseline to contextualize the post-quantum transition.

While the current single-accelerator implementation does not yet surpass software performance for short messages, the study provides a rigorous and reproducible methodology for performance evaluation and highlights clear directions for further optimization, such as parallel hashing, DMA integration, and higher-throughput interconnects.

This work contributes a comprehensive evaluation of LMS in an embedded secure boot context, combining algorithmic analysis, hardware–software co-design, and

quantitative benchmarking. The findings support the viability of LMS as a practical post-quantum signature algorithm for secure boot and establish a solid foundation for future research on efficient post-quantum cryptography in constrained environments.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The rapid progress of quantum computing represents a paradigm shift in the field of information security. Once large-scale, fault-tolerant quantum computers become available, they will be able to efficiently solve mathematical problems that underpin the security of classical public-key cryptography, such as integer factorization and discrete logarithms. Algorithms like RSA and ECDSA, which are widely deployed in secure boot and firmware authentication processes, would be rendered insecure, undermining the trustworthiness of critical embedded systems. This has prompted the cryptographic community to design, standardize, and deploy new post-quantum algorithms that remain secure even in the presence of quantum adversaries.
Embedded systems, and particularly automotive and industrial control units, face unique challenges in this transition. They operate under strict real-time constraints and limited computational and memory resources, yet their firmware must be authenticated during each boot to ensure integrity and authenticity. Replacing existing cryptographic primitives with post-quantum alternatives must therefore be carefully evaluated in terms of performance, memory footprint, and implementation complexity. In this context, selecting and integrating a quantum-safe digital signature scheme suitable for secure boot is a critical research objective.

1.2 Objectives and Scope

The main goal of this thesis is to evaluate the performance and practicality of standardized post-quantum digital signature algorithms within the secure boot process of real-time embedded systems. Specifically, the work focuses on:

- Reviewing the current state of post-quantum digital signature algorithms

standardized by NIST and the IETF, identifying suitable candidates for constrained embedded environments.

- Selecting the Leighton–Micali Signature (LMS) scheme as the primary focus, based on its strong security foundations, compact and constant key sizes, and straightforward hash-based construction.
- Implementing LMS in software on a Xilinx PYNQ-Z2 platform, adapting the reference code for reproducible single-core measurements.
- Integrating a SHA-256 hardware accelerator via an AXI-Lite interface to offload the hashing workload that dominates LMS signing and verification.
- Developing and optimizing a custom Linux device driver to minimize I/O latency and improve accelerator performance.
- Conducting a comprehensive benchmarking campaign, including execution-time analysis, cycle-level measurements, profiling, and comparison against a pre-quantum ECDSA baseline.

Through these steps, the thesis aims to characterize the performance of LMS in a realistic embedded setting and to explore how hardware–software co-design can mitigate the computational overhead of post-quantum schemes.

1.3 Methodology Overview

The methodology combines algorithmic analysis, embedded software implementation, FPGA-based hardware acceleration, and quantitative benchmarking:

1. **Algorithmic analysis**: Chapter 2 surveys post-quantum signature algorithms, comparing hash-based, lattice-based, and stateless schemes. Based on size, performance, and maturity criteria, LMS is selected as the focus.
2. **Software implementation**: Chapter 3 details the LMS and LM-OTS algorithms, parameter sets, and hash complexity. A reference implementation is adapted to run deterministically on the PYNQ-Z2.
3. **Hardware acceleration**: Chapter 4 describes the porting of a SHA-256 accelerator to the PYNQ fabric, its integration with the processing system, and the development of an optimized Linux driver using memory mapping to reduce syscall overhead.

4. **Benchmarking and analysis**: Chapter 5 presents the benchmarking methodology, including timing with C libraries, profiling with `gprof`, cycle-level analysis with `perf`, and break-even modeling. Results are compared with a pre-quantum ECDSA–SHA256 baseline.

This methodology provides both a rigorous performance evaluation of LMS on embedded hardware and a generalizable framework for analyzing other post-quantum algorithms.

1.4 Structure of the Thesis

The remainder of this document is organized as follows:

- Chapter 2 introduces the landscape of post-quantum digital signatures, comparing hash-based, lattice-based, and stateless schemes, and motivates the selection of LMS.
- Chapter 3 provides a detailed description of LMS and LM-OTS, including parameter sets, signing and verification procedures, and hash complexity analysis.
- Chapter 4 describes the hardware–software co-design, including the SHA-256 accelerator integration and driver optimization.
- Chapter 5 presents the benchmarking setup, methodology, and experimental results, including cycle-level analysis and comparison with an ECDSA baseline.
- Chapter 6 concludes the thesis, summarizing key findings and outlining directions for future work.

1.5 Contributions

The main contributions of this thesis are:

- A structured evaluation of standardized post-quantum signature schemes with a focus on their applicability to embedded secure boot.
- A reproducible LMS implementation on real hardware, accompanied by detailed hash complexity analysis.
- The design and optimization of a SHA-256 hardware accelerator integration for LMS on the PYNQ-Z2 platform.

- A comprehensive benchmarking and cycle-level characterization of LMS in both software-only and hardware-accelerated configurations.
- A break-even performance model for hardware acceleration and a quantitative comparison with classical ECDSA.

Together, these contributions provide practical insights into the deployment of LMS in resource-constrained secure boot scenarios and establish a foundation for further post-quantum hardware–software co-design research.

Chapter 2

Post-Quantum Digital Signatures

The modern and rapid technological evolution suggests that the development of a quantum computer, with enough qubits to break the strongest known encryption and digital signature algorithms, is imminent. This chapter explores the current state-of-the-art panorama to select a suitable digital signature algorithm to be implemented in the secure boot process.

2.1 State-of-the-art Overview

The design requires the algorithm to be computationally fast and space-efficient. With these requirements in mind, an analysis is performed on the standardized quantum-proof algorithms.

The National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization program in 2016 which led to the released final versions of the first three Post Quantum Crypto Standards: FIPS 203, FIPS 204, and FIPS 205. In addition there are already two known quantum-resistant digital signature algorithms: LMS and XMSS, described by the Internet Engineering Task Force (IETF) in RFC 8554 [1] and RFC 8391 [2]. An important innovation brought by the NIST program, is the use of new cryptographic families along with the already used ones: lattice-based and code-based cryptography and the multi-party computation paradigm. A performance review led by CloudFlare [3] analyses all the proposed schemes advanced to the second round. The results are briefly reported only selecting the current winners of the competition.

			Size [bytes]		CPU time	
Family	Algorithm Name	Quantum-Safe	Public Key	Signature	Signing	Verification
Elliptic Curves	Ed25519	No	32	64	0.15	1.3
Factoring	RSA 2048	No	256	256	80	0.4
Lattices	ML-DSA$_{44}$	Yes	1,312	2,420	1 (baseline)	1 (baseline)
	FALCON 512	Yes	897	666	3	0.7
Hash-based	SLH-DSA$_{128s}$	Yes	32	7,856	14,000	40
	SLH-DSA$_{128f}$	Yes	32	17,088	720	110
	LMS$_{M4_H20_W8}$	Yes	48	1,112	2.9	8.4

Table 2.1: Comparison of Digital Signature Algorithms

Table 2.1 shows a comparison between some selected variants of the standardized algorithms and earlier existing traditional and post-quantum schemes at the security level of AES-128. Values are taken from the Signature Zoo [4], times are relative to ML-DSA$_{44}$, since it is relatively balanced in terms of size, speed, and security. That makes it a good reference point to use for comparison. It is clear that the usage of post-quantum algorithms entails a loss of performance in CPU time and signature plus key total size and this is caused by the growing complexity of these algorithms in order to face quantum adversaries. CPU time varies significantly by platform and implementation, and these values were taken from the submission documents, so they should be taken with a grain of salt. A first look at the table highlights that SLH-DSA, also known as SPHINCS$^+$, is the worst performing algorithm, both in CPU time, signing alone performs 14,000 times worse than ML-DSA, and signature size. The current parameter sets of SHA2/SHAKE-128s and SHA2/SHAKE-128f, being the ones that provide the lowest security level, already reach significant signature size. Considering the highest security level parameter set of SHA2/SHAKE-256f it can reach sizes of 49856 bytes, which exceed the proposed design goals.

Algorithms based on lattices, on the other hand, appear to perform well. Lattice-based cryptography is a strong contender for post-quantum cryptography because it is resistant to attacks from both classical and quantum computers. In particular, FALCON (Fast-Fourier Lattice-based Compact Signatures over NTRU) shows a good space-time trade-off. It is in fact known for using fast Fourier sampling, which allows for very fast implementations. Its enhanced key generation algorithm uses less than 30 kilobytes of RAM, this makes it compatible with small, memory-constrained embedded devices.

LMS (Leighton-Micali Signature) also offers a reasonable space-time trade-off, especially when looking at the public key size. The values shown in Table 2.1 represent just one of the 40 parameter sets available for SHA2-based LMS. Therefore, a more in-depth analysis of the algorithm is necessary, starting with the fundamental concept of hash-based signatures.

2.2 Hash-based Digital Signatures

Hash-based signatures, such as SLH-DSA, LMS and XMSS, are a type of cryptographic signature scheme that relies on the properties of cryptographic hash functions for security. Their extensive history and proven security makes them highly reliable and well-understood. One key difference between hash-based signatures is them being stateless or stateful.

2.2.1 Stateful vs Stateless

Stateful hash-based schemes, such as LMS and XMSS, are based on one-time signature (OTS) schemes. Both algorithms use variants of the Winternitz OTS (WOTS).

Winternitz One-Time Signature

The Winternitz One-Time Signature (WOTS) scheme, unlike traditional signature schemes, uses a unique private key for each message, making it robust against key reuse and forgery attacks. In this scheme the private key consists of an array of random values, each serving as the starting point (seed) for a hash chain. For each element, a one-way function, typically a cryptographic hash function like SHA-256, is applied iteratively a fixed number of times, usually the maximum possible value that the integer can represent and the number of iterations is determined by the Winternitz parameter and the signing process. The public key is derived by hashing each private key element the maximum number of times and concatenating the final outputs.

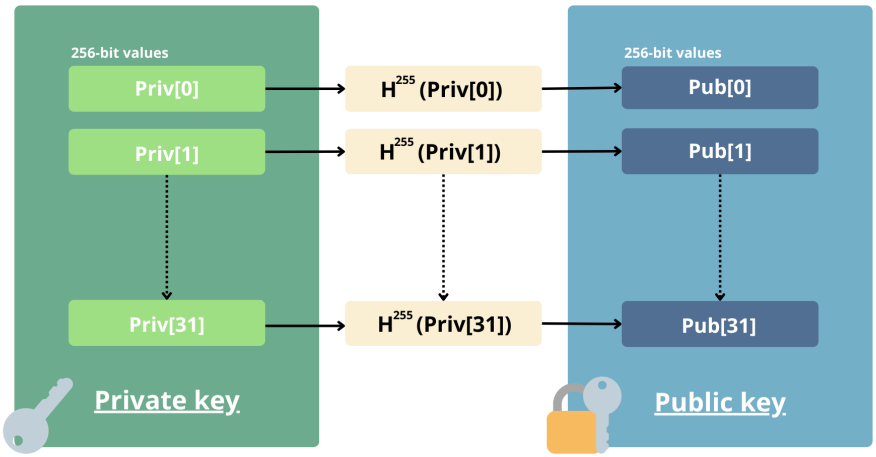


Figure 2.1: WOTS key generation

To sign a message using the WOTS scheme, the message digest is first split into chunks of $\log_2(w)$ bits, where w is the Winternitz parameter. Each chunk is interpreted as an integer value and used as an index into the private key array. For each chunk, the corresponding private key element is hashed a number of times equal to the value of the chunk.
To verify a WOTS signature, the verifier completes each hash chain by applying the hash function a number of times equal to the difference between the maximum chain length and the value of each corresponding chunk in the signature. The outputs of these completed chains form a public key candidate, which is then compared to the known public key. If they match, the signature is considered valid.

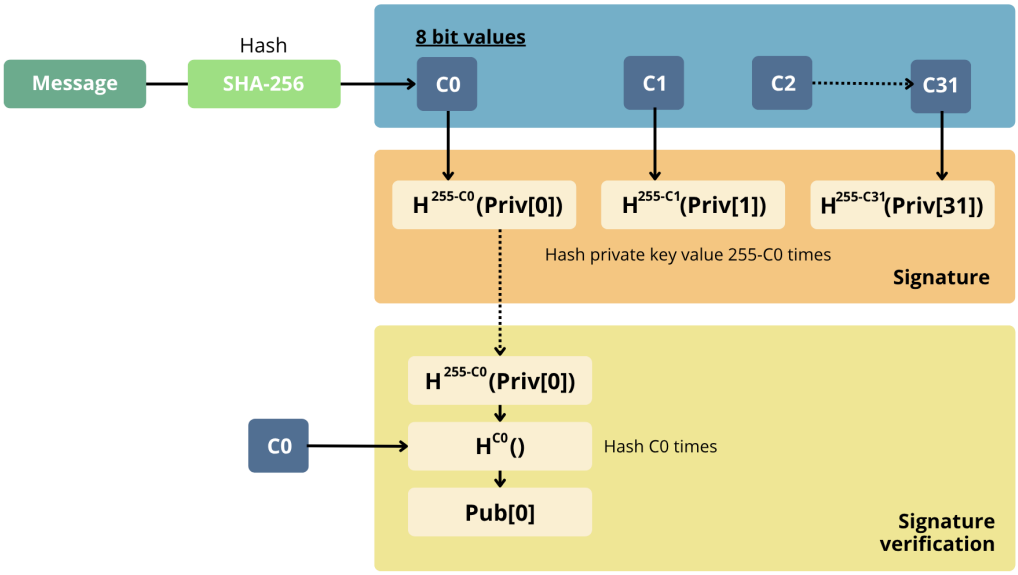


Figure 2.2: WOTS signature generation and verification

Merkle Signature Scheme

To overcome the limitation of one-time signatures, the Merkle Signature Scheme (MSS) is used. A Merkle tree authenticates 2^h OTS public keys where h is the height of the tree. Each leaf node in the tree is the hash of an OTS public key, and the root node of the tree serves as the MSS public key. The private key of the scheme consists of all 2^h OTS private/public key pairs (X_i, Y_i), as a result, the private key size grows linearly with the number of possible signatures.
During the signing process, the signature consists of the OTS signature of the message concatenated with the authentication path in the Merkle tree. The verifier uses this path, i.e. the intermediate nodes, to reconstruct the Merkle root and authenticate the OTS public key, ensuring it is valid and not forged. Once authenticated, the verifier proceeds to validate the message using the OTS verification algorithm. The security of the scheme is compromised if the same OTS key pair is used more than once, making it essential to carefully track which pairs have been used. Additionally, the finite number of available signatures must be taken into account. However, stateful hash-based signature schemes are well-suited for automotive systems, where infrequent updates make the limited number of signatures a manageable constraint.
On the other hand, stateless schemes like SLH-DSA do not require state management. This comes with the price of slower runtime and increased key size. A more in depth analysis of stateless schemes in a secure boot scenario can be found in [5].

Leighton-Micali Signatures (LMS)

Leighton-Micali Signature (LMS) is a hash-based, stateful digital signature scheme standardized by NIST and the IETF. The construction builds upon one-time signatures (OTS) arranged in a Merkle tree, allowing the generation of many signatures from a single public key, its security relies solely on the hardness of preimage and collision resistance of the underlying hash function, typically SHA-256 or SHAKE.

- The key generation process works as follows. The private key consists of a set of seeds used to derive private/public key pairs for the underlying OTS scheme (typically Winternitz OTS+). These OTS keys are organized in a binary Merkle tree of height h, where the root node represents the LMS public key. The public key is thus:

  $$pk = (I, T[1])$$

  where I is a unique identifier and $T[1]$ is the root of the Merkle tree.

- To sign a message M, the signer:

  1. Selects the next unused OTS key pair (sk_{OTS}, pk_{OTS}).
  2. Computes an OTS signature $\sigma_{OTS} = Sign_{OTS}(sk_{OTS}, H(M))$.
  3. Generates an authentication path consisting of the sibling nodes along the path from pk_{OTS} up to the Merkle root.
  4. Outputs the full LMS signature:

     $$\sigma' = (\sigma_{OTS}, pk_{OTS}, AuthPath)$$

  The scheme is stateful, since each OTS key can only be used once. The signer must maintain and update a counter to avoid key reuse, as reusing an OTS key compromises security.

- Given (M, σ', pk_{OTS}), verification proceeds as follows:

  1. Verify the OTS signature σ_{OTS} with pk_{OTS}.
  2. Recompute the Merkle tree path from pk_{OTS} using the authentication path.
  3. Accept if and only if the reconstructed root equals the public key root $T[1]$.

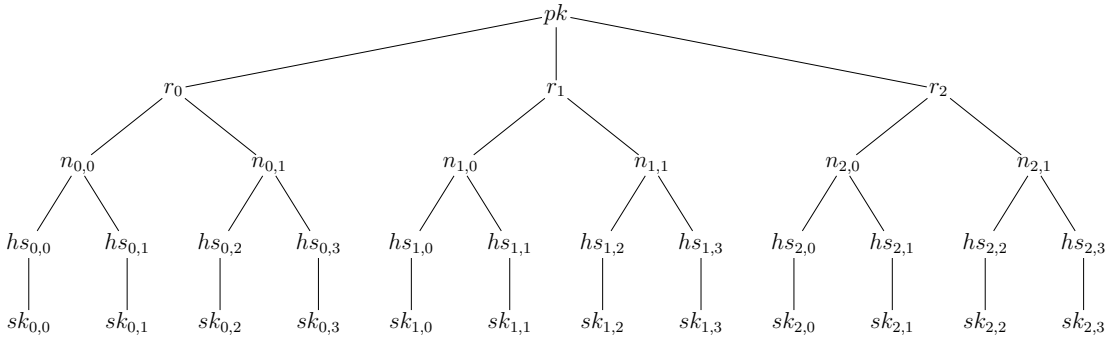
The main cost of LMS is computing the OTS signature and reconstructing the authentication path (which has length h). In terms of efficiency, LMS offers very compact key sizes: the public key is always 60 bytes, while the private key is fixed at 64 bytes. The main variability lies in the signature size, which depends on the choice of tree height h and Winternitz parameter w, as well as whether multiple trees are composed through HSS. Depending on these parameters, signature sizes can range from a few kilobytes to over 8.6 kilobytes. A detailed exploration of parameter sets and their impact on performance will be presented in Chapter 3, where LMS is analyzed in depth as the focus of this work.
Having discussed LMS as a representative of stateful hash-based signature schemes, it is now natural to contrast it with a stateless alternative. For this purpose, we turn to SLH-DSA, the leading candidate in the stateless category and the only stateless scheme selected by NIST for standardization. Before describing SLH-DSA itself, it is useful to outline its two core building blocks: FORS (Forest of Random Subsets) and hypertrees. These primitives extend the concepts of WOTS and Merkle trees introduced earlier and enable SLH-DSA to combine statelessness with post-quantum security.

Forest of Random Subsets (FORS)

A central building block of SLH-DSA is the Forest of Random Subsets (FORS), which is a few-time signature (FTS) scheme. Unlike one-time signature schemes such as WOTS, which can securely sign only a single message per key pair, FORS allows multiple signatures to be generated with the same private key before the security starts to degrade. This property avoids the key-reuse vulnerability of OTS while still offering strong post-quantum security.
At a high level, FORS is defined by three parameters:

- k: the number of trees,
- $t = 2^a$: the number of leaves in each tree,
- n: the security parameter (hash output length).

The private key consists of $k \cdot t$ secret strings of length n, derived from a single seed. The public key is created by hashing the k roots of each tree after these leaves are compressed to a root using a Merkle construction.
The structure for $k = 3$ and $t = 22$ is demonstrated in the example below:



Given a message digest of $k \cdot a$ bits, it is split into k substrings $l_0, \ldots, l_{k-1}$, each a bits long and interpreted as integers in $[0, t)$. Each l_i selects one secret key element sk_{i,l_i} from the i-th tree. The signature is formed by:

- revealing the k selected secret strings, one per tree,
- including the authentication paths that link them to the tree roots.

Verification consists of recomputing the k roots from the signature and hashing them to reconstruct the public key candidate and the signature is valid if it matches the known public key.
FORS can be used repeatedly without compromising security because each signature only discloses one secret string per tree, however, repeated usage gradually increases leakage, so the scheme balances efficiency and security. This is a big improvement over Lambport OTS, where security is compromised by a single reuse. A further advantage of FORS is its resilience to "weak messages," where parts of the digest repeat. In schemes such as Hash to Obtain Random Subset (HORS), which relies on a single large tree, repeated indices can lead to excessive leakage. In FORS, spreading indices across k independent trees mitigates this risk.

Hypertrees

Merkle trees are widely used in hash-based signatures to compress public keys: the root of the tree serves as a compact representation of many one-time or few-time public keys located at the leaves. However, when very large numbers of signatures must be supported, as required by NIST post-quantum standards (e.g., on the order of 2^{64} signatures per key pair), a single Merkle tree becomes impractical. The tree would need an enormous number of leaves, and both the construction and the transmission of authentication paths of length h would be expensive.
Hypertrees address this scalability issue by organizing multiple Merkle trees into a hierarchical structure. Instead of building one monolithic tree of height h, the tree is decomposed into d layers of smaller subtrees, each of height h/d. In this arrangement, the leaves of an upper-level tree are used to authenticate the roots of

lower-level trees. At the lowest level, the leaves are associated with one-time or few-time signature keys used to sign actual messages.
This layered construction has several advantages:

- It enables support for extremely large numbers of signatures without requiring a single massive Merkle tree.
- At signing time, only one subtree needs to be kept in memory, which significantly reduces storage requirements.
- Authentication paths consist of one short path per layer rather than a single very long path, reducing computational overhead.

SLH-DSA

SLH-DSA is a stateless hash-based signature scheme designed to provide post-quantum security without requiring state management, thereby eliminating the risk of key reuse errors. It constructs each signature entirely from freshly generated structures, allowing unlimited secure usage at the cost of larger signatures and slower signing times.
The introduction of FORS as a few-time signature scheme removes the need for signature state tracking, which was a limitation in earlier stateful schemes such as LMS and XMSS. This makes FORS a fundamental building block in the design of stateless schemes like SLH-DSA, which are required in the NIST post-quantum standardization process.
Combining all the cryptographic ingredients introduced previously, the high-level design of SLH-DSA can be described as follows. Given a message to be signed, it is first compressed into a digest of $k \cdot a$ bits using a hash function, according to the parameters of FORS. The choice of signing a digest rather than the message itself is necessary to guarantee the security of FORS, and it is also advantageous for performance and signature size. In particular, it ensures that the object being signed always has fixed and relatively small length. The resulting digest is signed using FORS, while the corresponding public key pk_{FORS} is then authenticated using a multi-tree construction based on XMSS^{MT} (XMSS with the use of hypertrees).
This latter part is itself modular: in XMSS^{MT}, the element to be signed is given as input to a WOTS+ instance, whose public key forms a leaf of a Merkle tree. These trees are then organized into a hypertree structure, where each root is signed by a leaf from the tree at the next higher level. At the top, the root of the hypertree represents the final public key of SLH-DSA. Thus, the SLH-DSA public key is compactly represented by the root of the hypertree, while the private key consists only of the seed material used to derive the FORS and WOTS+ keys that populate the various structures. This leads to very small public and private key sizes, which is a notable strength of SLH-DSA.

Finally, a SLH-DSA signature on a message is the concatenation of:

- the FORS signature of the message digest,
- the WOTS+ signatures authenticating the FORS public key and the roots of the Merkle trees,
- the authentication paths required to validate the nodes within the hypertree.

This composition creates a stateless design that balances strong security with adaptability. Even though signatures are large and signing is expensive in terms of computing power compared to stateful alternatives like LMS or XMSS, SLH-DSA is very strong and easier to implement correctly in practice because it doesn't require state management.

2.3 Lattice-based Digital Signatures

Lattice-based digital signatures security relies on well-studied mathematical problems conjectured to be hard even for quantum computers. Lattice-based schemes offer reusable keys, relatively small signature sizes, and fast verification.

2.3.1 Introduction to Lattice-Based Cryptography

Lattices are discrete, regularly spaced sets of points in a multi-dimensional space defined by a basis, a set of linearly independent vectors that span the lattice. Digital signatures schemes are based on several hard problems related to lattices. Two prominent examples are the Learning with Errors (LWE) problem and the Short Integer Solution (SIS) problem.

Learning With Errors Problem

This problem is based on representing secret information as a system of linear equations that has been noised by small random errors. As the dimension grows, the presence of errors makes the problem extremely difficult. The LWE problem has been used as a computational hardness assumption to create public key cryptosystems like the ring learning with errors key exchange (RLWE-KEX). The RLWE problem is the broader LWE problem focused on polynomial rings over finite fields, which is believed to be resistant against quantum computers. A significant advantage of RLWE-based cryptography over LWE lies in the size of the public and private keys. RLWE schemes drastically reduce key sized to around the square root of those used in LWE. Despite this improvement, RLWE key sizes are still larger than those used in common public-key algorithms, such as RSA

(3,072 bits) and Elliptic Curve Diffie-Hellman (256 bits) at the same security level. Nonetheless, RLWE-based cryptographic schemes offer competitive or even superior computational performance compared to these classical systems.

Short Integer Solution Problem

The goal of this problem is to find a short non-zero vector that satisfies a particular linear relation. Although many solutions to this problem may exist, finding one with the required constraint of the length of the solution is computationally hard. Beyond their role in classical cryptography, the SIS problem and its variants form the foundation of several post-quantum cryptographic schemes, including ML-DSA (formerly called CRYSTALS-Dilithium) and FALCON.

2.3.2 Overview of Lattice-Based Signature Schemes

While hash-based signature schemes are discussed separately, it's important to note that many lattice-based schemes also rely heavily on cryptographic hash function. Lattice-based digital signature schemes can be broadly classified into two design paradigms: *hash-and-sign* and *Fiat-Shamir*.

Hash-and-Sign signatures

Signatures based on hash functions are among the simplest and fastest post-quantum alternatives. However, their core property, collision-resistance, can be attacked by quantum computers. Grover's algorithm offers a quadratic speed-up for searching for preimages and collisions in hash functions. To counteract this, one can increase the output size of the hash function, for example, by using SHA-512 and SHA-3, thus maintaining an adequate level of security, since the increase in size counteracts the quadratic speed-up. The most well-known instantiation of the hash-and-sign paradigm in lattice-based cryptography is the GPV framework [6]. This framework relies on the notion of a trapdoor one-way function, namely a function $f_A(x) = A \cdot x \pmod{q}$ that is easy to evaluate but hard to invert without additional information. Here, the matrix $A \in (\mathbb{Z}_q)^{nxm}$ serves as the public key while the trapdoor is a short basis of the lattice associated with A. which enables efficient sampling of short preimages x from discrete Gaussian distributions such that $A \cdot x \equiv y(\pmod{q})$ for a given target y. In the signing procedure, the message is first hashed to a syndrome $h(m)$, and the signer uses the trapdoor to compute a short vector x with $A \cdot x \equiv h(m) \pmod{q}$. The signature consists of this short preimage, while verification only requires checking the congruence and the norm bound. The hardness of finding such short solutions without the trapdoor ensures the unforgeability of the scheme.

FALCON

A remarkable realization of the GPV framework is the NIST Post-Quantum Cryptography finalist FALCON signature scheme. It instantiates the GPV paradigm using the NTRU lattice and introduces Fast Fourier sampling as an efficient trapdoor sampling method. The underlying hard problem is the SIS problem over NTRU lattices, which has no known efficient solving algorithm in the general case. The NTRU problem is the following: given n a power of 2 and q a prime number, a set of NTRU secret is composed of 4 small polynomials

$$f, g, F, G \in \mathbb{Z}[X]/(X^n + 1)$$

such that

$$fG - gF = q \pmod{X^n + 1}$$

with f invertible modulus q.
In this scenario, the secret key is (f, g, F, G) and the public key is the polynomial

$$h = g \cdot f^{-1} \pmod{q}$$

The NTRU problem, considered hard even for quantum computers, consists in finding two polynomials with small coefficients f', g' such that $h \equiv g'f'^{-1} \pmod{q}$. This problem is closely related to lattice theory, since the best known cryptanalytic algorithm reduces to solving an instance of the Shortest Vector Problem (SVP) in the lattice generated by the $2n \times 2n$ integer matrix.

$$\begin{bmatrix} 1 & h \\ 0 & q \end{bmatrix} := \left(\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & h_0 & h_1 & \cdots & h_{n-1} \\ 0 & 1 & \cdots & 0 & -h_{n-1} & h_0 & \cdots & h_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & -h_1 & -h_2 & \cdots & h_0 \\ \hline 0 & 0 & \cdots & 0 & q & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & q & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & q \end{array}\right)$$

where $h = h_0 + h_1X + \ldots + h_{n-1}X^{n-1}$.
FALCON's main design principles are compactness to minimize $|pk| + |sig|$ and speed.
It operates over the cyclotomic ring $\mathcal{R} = \mathbb{Z}_q[x]/(x^n + 1)$.

- Key generation works as follows:

1. Generates two matrices A and B with coefficients in $\mathcal{R}$ such that $BA = 0$ and B has small coefficients. More specifically, instantiate the GPV framework with the following parameters:

$$A = \begin{bmatrix} 1 & h^* \end{bmatrix} \in \mathbb{Z}_q^{n \times 2n}$$

$$B = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix} \in \mathbb{Z}_q^{n \times 2n}$$

where $h^* = h_0 + h_{n-1}X + h_{n-2}X^2 + ... + h_1X^{n-1}$ and it's possible to verify that

$$BA^* = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix} \begin{bmatrix} 1 \\ h \end{bmatrix} = 0 \pmod{q}$$

2. pk $\leftarrow A$
3. sk $\leftarrow B$

- The signature $Sign(m, sk)$ process is:

1. A random salt r is generated to ensure security, preventing two distinct valid signatures for the same hash $H(m)$. The message is then hashed as $H(r\|m)$ using SHAKE-256.
2. Compute a target c such that $cA = H(r\|m)$.
3. Use Fast Fourier Sampling to find a short vector $v \in \Lambda(B)$ close to c.
4. Set $s \leftarrow c - v$, where $s = (s_1, s_2)$ are polynomials satisfying the relation

$$s_1 + s_2h^* \equiv H(r\|m) \pmod{q}.$$

The signature is the pair $(r, (s_1, s_2))$.

- The verification $Verify(m, pk, sig)$ process is:

1. Parse the signature as $(r, (s_1, s_2))$.
2. Recompute the target hash $H(r\|m)$ using SHAKE-256.
3. Check the algebraic relation

$$s_1 + s_2h^* \equiv H(r\|m) \pmod{q}.$$

4. Verify that the vector $s = (s_1, s_2)$ is short.

The signature is accepted if and only if both conditions are satisfied.

Falcon is fast and the most compact of all post-quantum signature schemes. Parameters and performance of Table2.2, according to [7], are taken on an Intel Skylake @ 3.3Ghz.

NIST level	n	q	$\|pk\|$ (bytes)	$\|sig\|$ (bytes)	Sign/sec.	Verify/sec.
1	512	$12 \cdot 1024 + 1$	897	618	6082	37175
4-5	1024	$12 \cdot 1024 + 1$	1793	1233	3073	17697

Table 2.2: FALCON parameters and performances

It is worth noting that the signature sizes reported in Table 2.2 (e.g., 618 bytes at NIST level 1) refer to the compressed representation. In practice, implementations use a fixed-length padded format of 666 bytes to simplify parsing and maintain constant-time behavior. Therefore, while the compressed size is useful for performance estimates, the standardized size is slightly larger.
Table 2.3, adapted from [8], presents performance benchmarks measured on an Intel Core i5-8259U @ 2.3 GHz with TurboBoost disabled. The reported RAM usage corresponds to the key generation process and is expressed in bytes, while public key and signature sizes are given in their standardized, uncompressed form.

Variant	keygen (ms)	keygen (RAM)	$\|pk\|$ (bytes)	$\|sig\|$ (bytes)	Sign/sec.	Verify/sec.
FALCON-512	8.64	14336	897	666	5948.1	27933.0
FALCON-1024	27.45	28672	1793	1280	2913.0	13650.0

Table 2.3: FALCON parameters and performances

Falcon stands out for its compact signatures and high verification speed, which make it attractive for bandwidth-constrained environments and large-scale systems. Its enhanced key generation algorithm requires less than 30 KB of RAM, a major improvement over earlier lattice-based designs such as NTRUSign, suggesting compatibility with embedded devices. However, the reliance on floating-point arithmetic in the signing process complicates secure implementations, particularly on constrained hardware, since care must be taken to avoid precision errors, side-channel leakage, and higher working memory demands. For this reason, while Falcon was standardized by NIST alongside ML-DSA, the latter is generally preferred in implementations prioritizing robustness and ease of deployment. Falcon, nevertheless, remains the most compact lattice-based signature scheme available and a critical component of the post-quantum cryptography landscape.

ML-DSA

ML-DSA represents the Fiat–Shamir paradigm, prioritizing implementation simplicity and robustness over compactness. In particular, it is based on the Fiat–Shamir with Aborts technique [9], which employs rejection sampling to ensure security. Unlike schemes such as Falcon that rely on discrete Gaussian sampling, ML-DSA instead adopts uniform sampling over small coefficients. Combined with its module lattice construction, this approach avoids the implementation challenges of discrete Gaussian sampling and achieves a reduction of the public key size by roughly a factor of 2.5 compared to earlier lattice-based signature schemes, while preserving both the security guarantees and the signature length.

- Key generation proceeds as follows:

  1. Sample uniformly a public matrix $A \in R_q^{k \times l}$, where each entry is a polynomial in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$.
  2. Sample the secret key components (s_1, s_2) uniformly from S_η, a subset of R_q with coefficients limited by η.
  3. Compute $t = A \cdot s_1 + s_2 \in R_q^k$.
  4. The public key is (A, t), while the secret key is (s_1, s_2).

  The security of Dilithium relies on the hardness of the Module-LWE problem: given (A, t), recovering the secrets (s_1, s_2) is as hard as solving an instance of Module-LWE.

  As mentioned above, Dilithium samples (s_1, s_2) uniformly. This choice simplifies implementations and reduces susceptibility to certain side-channel attacks.

  Since the matrix A consists of $k \cdot l$ polynomials in R_q, storing and transmitting it directly would be impractical. Instead, A is deterministically generated from a short random seed, meaning only the seed needs to be included in the public key.

- Signing $Sign(m, sk)$ works as follows:

  1. Sample a masking vector $y \in R_q^l$ with coefficients bounded by γ_1. The parameter γ_1 is chosen large enough to prevent the signature from revealing the secret key, yet small enough to prevent easy forgery.
  2. Compute $w = Ay$ and decompose each coefficient as

  $$w = w_1 \cdot 2^{\gamma_2} + w_0, \quad |w_0| \leq \gamma_2,$$

  where w_1 collects the high-order bits of w.

3. Generate the challenge polynomial $c \leftarrow H(m \parallel w_1)$, where c has exactly 60 coefficients equal to ± 1 and the rest zero.
4. Compute the candidate signature vector:

$$z = y + c \cdot s_1.$$

5. Apply rejection sampling, with β defined as the maximum coefficient of cs_i:
   - If any coefficient of z exceeds $\gamma_1 - \beta$, reject and restart.
   - If any coefficient of the low-order part of $Az - ct$ exceeds $\gamma_2 - \beta$, reject and restart.

The parameters are chosen such that the expected number of repetitions of the while loop is between 4 and 7. The rejection sampling step ensures that the final signature (z, c) does not leak information about the secret key, preserving zero-knowledge security.

- Verification $Verify(m, pk, sig)$ works as follows:

1. Given the signature (z, c), compute

$$w' = Az - ct,$$

   and set $w_1' = \mathsf{HighBits}(w', 2\gamma_2)$.
2. Check the following conditions:
   - All coefficients of z are bounded by $\gamma_1 - \beta$.
   - The challenge c is equal to $H(m \parallel w_1')$.
3. Accept if and only if both checks hold.

Correctness follows from the observation that

$$w' = Az - ct = A(y + cs_1) - c(As_1 + s_2) = Ay - cs_2.$$

Thus,

$$\mathsf{HighBits}(Ay - cs_2, 2\gamma_2) = \mathsf{HighBits}(Ay, 2\gamma_2),$$

since the term cs_2 only affects the low-order part of the coefficients. Because a valid signature satisfies

$$\|\mathsf{LowBits}(Ay - cs_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta,$$

the addition of cs_2 cannot cause carries into the high-order bits. Therefore the verification equation holds, ensuring correctness.

The main goal of the authors of ML-DSA is to build a scheme that minimizes public key and signature sizes. Table 2.4, adapted from [10], reports the performance of key generation, signing, and verification, expressed in CPU clock cycles, as measured on an Intel Core-i7 6600U (Skylake) processor.

NIST level	Keygen	Sign	Verify	$\|pk\|$ (bytes)	$\|sig\|$ (bytes)
2	300751	1355434	327362	1312	2420
3	544232	2348703	522267	1952	3293
5	819475	2856803	871609	2592	4595

Table 2.4: ML-DSA performances

In conclusion, ML-DSA prioritizes robustness and deployment simplicity, avoiding the complications of Gaussian sampling by using uniform sampling and the Fiat–Shamir with Aborts framework. In contrast to previous lattice-based schemes, its design offers smaller public keys and robust security guarantees based on the Module-LWE assumption. Larger signature sizes and a dependence on rejection sampling, which can result in some computational overhead, are the price paid for these advantages. However, ML-DSA is a sensible and well-balanced option, and its standardization with Falcon emphasizes its significance as a fundamental component of post-quantum digital signature systems.

2.4 Comparative Analysis

The purpose of this section is to compare representative post-quantum digital signature algorithms across the main candidate families, with the goal of identifying the most suitable scheme for secure boot in automotive embedded systems. The analysis focuses on three primary quantitative metrics: public key size, private key size, and signature size. These parameters are of particular importance in constrained environments, where memory footprint and communication overhead directly impact feasibility.

It should be noted that performance in terms of signing and verification speed is highly implementation and platform dependent. A fully objective comparison would require testing each algorithm under identical hardware and software conditions, which is beyond the scope of this work. Instead, we rely on reported values from the NIST submissions and related literature, while acknowledging that such figures should be interpreted with caution.

In addition to raw performance characteristics, it is also necessary to consider the maturity and deployability of each algorithm. For example, the Commercial

National Security Algorithm Suite (CNSA) 2.0 [11] published by the U.S. National Security Agency explicitly recommends the use of LMS and HSS for code signing and firmware updates, reflecting their practicality and robustness in real-world applications. More generally, hash-based signatures benefit from decades of study and simplicity of design, whereas lattice-based schemes, while compact and efficient, are comparatively recent and require more complex implementations. These considerations guide the final choice of the candidate algorithm to be evaluated in this thesis.

2.4.1 Quantitative Comparison

As shown in Table 2.5, all considered schemes feature compact public and private keys, generally on the order of a few dozen bytes (for hash-based schemes) to a few kilobytes (for lattice-based schemes). The dominant factor in storage and transmission overhead is therefore the signature size. Falcon achieves the smallest signatures, below 1.3 KB even at the highest security level, while SLH-DSA exhibits the largest, ranging up to nearly 50 KB. LMS lies in the middle, with signatures between 1 and 9 KB, and ML-DSA provides moderate values around 2–5 KB. This variability in signature footprint is one of the most important distinguishing factors between the schemes.

Scheme	Public Key Size	Private Key Size	Signature Size (Range)
LMS	60 B	64 B	1.3 – 9.2 KB
SLH-DSA (SPHINCS+)	32 – 64 B	32 – 96 B	7.9 – 49.9 KB
FALCON	897 – 1,793 B	~2 KB	666 – 1,280 B
ML-DSA (Dilithium)	1.3 – 2.6 KB	2.5 – 4.9 KB	2.4 – 4.6 KB

Table 2.5: Comparison of key and signature sizes across PQ signature schemes.

To help show these differences, Figure 2.3 provides a bar chart of signature sizes with ranges across parameter sets. It highlights how compact Falcon is, the moderate impact of LMS and ML-DSA, and the large signatures of SLH-DSA.

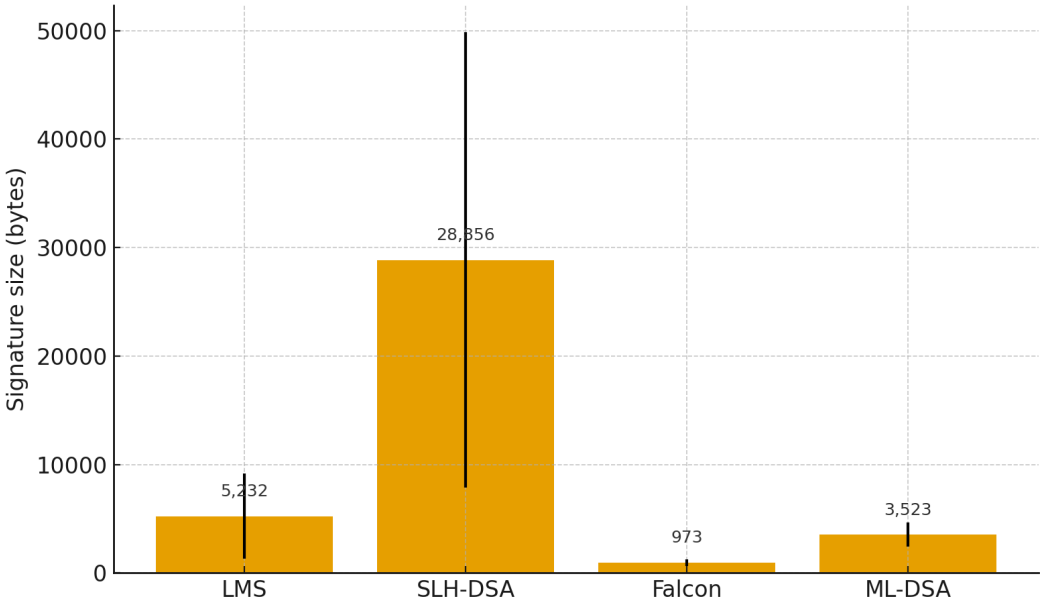


Figure 2.3: Signature Sizes of PQ Signature Schemes (Range Across Parameters)

From this comparison it is evident that signature size is the primary factor influencing the practicality of post-quantum digital signature schemes. In constrained environments such as embedded devices or automotive control units, large signatures directly translate into higher memory requirements and increased communication overhead during software updates or secure boot. This is particularly problematic for schemes like SLH-DSA, whose signatures can reach tens of kilobytes, making them less suitable for bandwidth-constrained or latency-sensitive deployments.
Falcon, on the other hand, offers the smallest signatures among the candidates, which makes it attractive for scenarios where communication efficiency is paramount. However, its reliance on floating-point arithmetic and Gaussian sampling complicates secure and side-channel resistant implementations, especially on resource-limited platforms. ML-DSA represents a more balanced lattice-based alternative, with moderately larger signatures than Falcon but simpler and more robust implementation requirements. Nevertheless, both lattice-based families remain relatively recent and rely on complex mathematical assumptions, in contrast to the well-established security foundations of hash-based constructions.
Hash-based schemes, and LMS in particular, present a compelling trade-off. Signature sizes are larger than those of Falcon or ML-DSA, but remain within a practical range and scale predictably with chosen parameters. Key sizes are minimal and constant, which simplifies storage. Most importantly, LMS benefits from decades of study, simple cryptographic assumptions, and strong standardization support.
In conclusion, while lattice-based schemes such as Falcon and ML-DSA offer

compact signatures and strong performance, their complexity and relative novelty introduce challenges for deployment in constrained environments. Stateless hash-based signatures like SLH-DSA provide robust security guarantees without state management, but their very large signatures render them impractical for secure boot and firmware signing. By contrast, LMS achieves a favorable balance of simplicity, moderate signature sizes, minimal key material, and strong standardization support. For these reasons, this work selects LMS as the candidate post-quantum digital signature algorithm to be evaluated in depth, with a focus on its integration into a secure boot framework for embedded systems.

Chapter 3

Leighton–Micali Signatures (LMS)

3.1 Introduction

Leighton–Micali Signatures (LMS) are stateful, hash-based signature schemes that combine one-time signatures with a Merkle tree to enable multiple authenticated signatures under a single public key. Standardized by the IETF (RFC 8554), LMS represents one of the most mature and conservative post-quantum signature algorithms available today. Its security rests only on the properties of the underlying hash function, making it well-suited for applications such as firmware and secure boot, where simplicity, robustness, and long-term trustworthiness are critical.
The goal of this chapter is to provide a detailed and implementation-oriented analysis of LMS, both to establish a clear understanding of the algorithm and to motivate the methodology of this thesis, namely the acceleration of hash operations in secure boot. To this end, the chapter is organized as follows:

- Section 3.2 introduces the notation and conventions used throughout the chapter.
- Section 3.3 presents the Leighton–Micali One-Time Signature (LM-OTS) scheme, describing its parameter sets, key generation, signing, and verification procedures.
- Section 3.4 builds on LM-OTS to describe the full LMS construction, including private and public key generation, signing, and verification.
- Section 3.5 quantifies the number of hash function invocations required for each operation across parameter sets, highlighting the computational cost.

- Section 3.6 discusses optimizations such as hierarchical signatures (HSS) and their implications for practical deployments.

This structure first establishes a clear and rigorous description of LMS, then provides evidence that accelerating hash operations is a justified and necessary methodology for secure boot scenarios.

3.2 Notation and Assumptions

We adopt the following notation and conventions throughout this chapter. Parameters and variables are defined here once and reused consistently in the descriptions of LM-OTS and LMS.

- $H : \{0,1\}^* \to \{0,1\}^n$ — a cryptographic hash function with output length n bytes (typically $n = 32$ for SHA-256).
- I — a fixed identifier associated with a key pair, included in all hash computations to prevent multi-target forgeries.
- q — the leaf index, representing which LM-OTS key pair is used within the Merkle tree.
- h — the height of the Merkle tree, which determines the total number of one-time key pairs: 2^h.
- $w \in \{1,2,4,8\}$ — the Winternitz parameter for LM-OTS. Larger w values yield shorter signatures but require more hash computations during signing and verification.
- n — the security parameter, equal to the output length of the hash function in bytes.
- p — the number of elements in the LM-OTS private key. This depends on n and w and is formally defined in Section 3.3.
- $\mathrm{PRF}(K, \cdot)$ — pseudorandom function derived from H used to expand a master seed K into per-leaf secret values.
- $\mathsf{DS}_{(\cdot)}$ — domain-separation tags used for each type of hash call (e.g., node computation, OTS chain step, public key hashing).
- pk, sk — public key and secret key, respectively.
- $\|$ — denotes string concatenation.

We assume constant-time implementations of H and reliable state management for LMS, meaning that the system prevents reuse of the same LM-OTS private key. This is critical to ensure the unforgeability of the scheme.

3.3 LM-OTS One-Time Signatures

LM-OTS (Leighton–Micali One-Time Signature) is the basic building block of LMS. Each LM-OTS private key may only be used once to sign a message, therefore reusing the same key for multiple messages would immediately break security. The scheme associates a secret private key with a corresponding public key, and signatures are generated over a digest of the message.

3.3.1 Parameters

The LM-OTS scheme relies on the general parameters introduced in Section 3.2, specialized to the context of one-time signatures:

- The security parameter n defines the output length of the hash function H. For instance, with SHA-256, $n = 32$ bytes.
- The Winternitz parameter $w \in \{1,2,4,8\}$ governs the trade-off between signature size and computational cost. Larger values of w result in smaller signatures but require more hashing during signing and verification.
- The value of p, the number of n-byte string elements in a signature, is derived from n and w.
- The parameter ls is the left-shift amount used in the checksum function. While it does not affect performance directly, it plays a role in ensuring the integrity of the scheme.

In summary, w controls the space–time trade-off of LM-OTS: increasing w reduces the signature size (since p decreases) but makes each signature more computationally demanding.
The overall LM-OTS signature size is

$$|sig| = 4 + n \cdot (p + 1) \quad \text{bytes},$$

where the extra n accounts for the randomizer C used during signing.
Table 3.1 summarizes the parameter sets standardized in RFC 8554 [1], all instantiated with SHA-256.

Name	Hash	n	w	p	ls	$\|sig\|$ (bytes)
LMOTS_SHA256_N32_W1	SHA-256	32	1	265	7	8516
LMOTS_SHA256_N32_W2	SHA-256	32	2	133	6	4292
LMOTS_SHA256_N32_W4	SHA-256	32	4	67	4	2180
LMOTS_SHA256_N32_W8	SHA-256	32	8	34	0	1124

Table 3.1: LM-OTS parameter sets with SHA-256 [1].

For example, with $w = 1$ the signatures are very large (8.5 KB) but fast to generate and verify, while $w = 8$ returns much shorter signatures (about 1.1 KB) but at higher computational cost.

3.3.2 Private Key Generation

An LM-OTS private key is associated with a specific leaf of an LMS Merkle tree, identified by the pair (I, q) where:

- I is a 16-byte identifier unique to the LMS key pair,
- q is the 32-bit index of the leaf node within the tree.

These identifiers are included in all hash computations to guarantee uniqueness and prevent cross-protocol forgeries.
The private key itself consists of an array of p secret values $x[0], x[1], \ldots, x[p-1]$, each of length n bytes. These values must be generated uniformly at random, or equivalently, they may be derived pseudorandomly from a master seed using a pseudorandom function.
Formally, private key generation proceeds as follows:

1. Retrieve the identifiers (I, q) from the LMS tree.
2. Fix the LM-OTS parameter set (n, w, p, ls) according to Table 3.1.
3. For each $i \in \{0, \ldots, p-1\}$, generate a random n-byte string $x[i]$.

The private key is the tuple:

$$\mathsf{sk} = (\mathtt{type}, I, q, x[0], x[1], \ldots, x[p-1]),$$

where `type` encodes the chosen LM-OTS parameter set.
In practice, storing p independent random strings is expensive. For this reason, implementations often derive each $x[i]$ pseudorandomly from a short seed K using a PRF:

$$x[i] = \mathrm{PRF}(K, I \parallel q \parallel i).$$

This approach reduces private key storage to a single n-byte seed, without weakening security, provided the PRF is cryptographically strong.

3.3.3 Public Key Generation

The LM-OTS public key is derived from the private key by applying the hash function H iteratively to each element of the private key array. Each private key element $x[i]$ is the starting point of a Winternitz chain, which is advanced $2^w - 1$ steps by repeated hashing. The final outputs of all chains are then hashed together to form the public key.
The process is as follows:

1. Retrieve the parameter set (n, w, p, ls) and the private key elements $x[0], \ldots, x[p-1]$.
2. For each $i \in \{0, \ldots, p-1\}$:
   (a) Set $tmp \leftarrow x[i]$.
   (b) For $j = 0$ to $2^w - 2$:
   $$tmp \leftarrow H(I \parallel u32(q) \parallel u16(i) \parallel u8(j) \parallel tmp),$$
   where $u32$, $u16$, and $u8$ denote fixed-length encodings of integers.
   (c) Store the final value $y[i] \leftarrow tmp$.
3. Compute the public key root value:
   $$K = H(I \parallel u32(q) \parallel u16(D_{\mathtt{PBLC}}) \parallel y[0] \parallel y[1] \parallel \cdots \parallel y[p-1]),$$
   where $D_{\mathtt{PBLC}} = \mathtt{0x8080}$ is a domain-separation constant.

The LM-OTS public key is then defined as:

$$\mathsf{pk} = (\mathtt{type}, I, q, K).$$

Intuitively, the public key binds all the private key chains into a single n-byte digest K. Any signature will consist of intermediate points on these chains, allowing the verifier to recompute the end points $y[i]$ and check consistency with K.
The LM-OTS algorithm includes a checksum over the message digest, which ensures that attempts to manipulate Winternitz chains are detected. The checksum's computation involves only simple integer additions and shifts, with negligible performance impact compared to the hash operations. For this reason, it is not analyzed further in this work.

3.3.4 Signature Generation

To sign a message M, the LM-OTS signer first hashes the message into a digest Q, appends a checksum, and then encodes the result as a sequence of base-2^w digits. Each digit determines how far to advance the corresponding Winternitz chain from the private key. The intermediate values reached along the chains form the signature.
The signing procedure is as follows:

1. Retrieve the parameter set (n, w, p, ls), the private key elements $x[0], \ldots, x[p-1]$, and the identifiers (I, q).
2. Choose a random n-byte string C.
3. Compute the message digest:

$$Q = H(I \parallel u32(q) \parallel u16(D_{\mathsf{MESG}}) \parallel C \parallel M),$$

   where $D_{\mathsf{MESG}} = \texttt{0x8181}$ is a domain-separation constant.
4. Concatenate Q with its checksum $\mathsf{Cksm}(Q)$, and parse the result into p w-bit coefficients $a_0, a_1, \ldots, a_{p-1}$.
5. For each $i \in \{0, \ldots, p-1\}$:
   (a) Set $tmp \leftarrow x[i]$.
   (b) Apply the hash function a_i times:

$$tmp \leftarrow H(I \parallel u32(q) \parallel u16(i) \parallel u8(j) \parallel tmp),$$

   for $j = 0, \ldots, a_i - 1$.
   (c) Output $y[i] \leftarrow tmp$.
6. The signature is the tuple:

$$\sigma = (\texttt{type}, C, y[0], y[1], \ldots, y[p-1]).$$

Each $y[i]$ is therefore an intermediate point along the i-th Winternitz chain that, together with the coefficients a_i, allow the verifier to complete the chains up to their final values $y[i]^{(2^w-1)}$ and check consistency with the public key K.

3.3.5 Signature Verification

Verification of an LM-OTS signature consists of recomputing the end of each Winternitz chain from the signature elements $y[i]$, and checking that the resulting public key candidate matches the known public key K.
The procedure is as follows:

1. Parse the signature $\sigma = (\mathtt{type}, C, y[0], \ldots, y[p-1])$, along with the identifiers (I, q) and public key K.
2. Recompute the message digest:
$$Q = H(I \parallel u32(q) \parallel u16(D_{\mathsf{MESG}}) \parallel C \parallel M).$$
3. Concatenate Q with its checksum $\mathsf{Cksm}(Q)$, and parse the result into p w-bit coefficients $a_0, a_1, \ldots, a_{p-1}$.
4. For each $i \in \{0, \ldots, p-1\}$:
   (a) Set $tmp \leftarrow y[i]$.
   (b) Apply the hash function $(2^w - 1 - a_i)$ times:
   $$tmp \leftarrow H(I \parallel u32(q) \parallel u16(i) \parallel u8(j) \parallel tmp),$$
   for $j = a_i, \ldots, 2^w - 2$.
   (c) Store the final value $z[i] \leftarrow tmp$.
5. Compute the public key candidate:
$$K_c = H(I \parallel u32(q) \parallel u16(D_{\mathsf{PBLC}}) \parallel z[0] \parallel z[1] \parallel \cdots \parallel z[p-1]).$$
6. Accept the signature if and only if $K_c = K$.

Intuitively, verification completes each Winternitz chain from the point published in the signature, ensuring that the revealed values are consistent with the public key digest K. Additionally, the checksum enforces that no coefficients can be freely increased, which results in a detection of any forgery attempt.

3.4 Leighton–Micali Signatures

While LM-OTS allows only a single message to be signed securely with one key pair, the Leighton–Micali Signature scheme extends this construction by authenticating a large set of LM-OTS keys using a Merkle tree. This allows many signatures to be produced under a single LMS public key, making the scheme practical for deployment.

3.4.1 Parameters

LMS is defined by the following parameters:

- h — the height of the Merkle tree. The tree authenticates 2^h LM-OTS key pairs.
- `LM-OTS_type` — the LM-OTS parameter set used for the leaves. For example, LMOTS_SHA256_N32_W8.
- H — the hash function used throughout the scheme (typically SHA-256 with $n = 32$).
- I — the 16-byte identifier unique to the LMS key pair.

The overall security level of LMS depends jointly on the security of the underlying hash function and the chosen LM-OTS parameter set.

3.4.2 Private Key Generation

The LMS private key consists of the complete set of LM-OTS private keys for all 2^h leaves of the Merkle tree. Each leaf corresponds to one LM-OTS key pair identified by its leaf index q.
Private key generation proceeds as follows:

1. Fix the LMS parameters $(h, \texttt{LM-OTS\_type}, H)$.
2. For each leaf index $q \in \{0, \ldots, 2^h - 1\}$:
   (a) Generate the LM-OTS private key

$$\mathsf{sk}_{\mathrm{OTS}}^{(q)} = (\texttt{LM-OTS\_type}, I, q, x[0], \ldots, x[p-1]),$$

   as described in Section 3.3.

Thus the full LMS private key is the collection:

$$\mathsf{sk}_{\mathrm{LMS}} = (I, h, \texttt{LM-OTS\_type}, \{\mathsf{sk}_{\mathrm{OTS}}^{(q)}\}_{q=0}^{2^h-1}).$$

In practice, storing all 2^h LM-OTS private keys explicitly is inefficient. Instead, implementations typically store only a master seed and derive the necessary LM-OTS private keys pseudorandomly on demand using a PRF, in the same way that LM-OTS elements $x[i]$ can be derived from a seed.

3.4.3 Public Key Generation

The LMS public key is derived by authenticating all 2^h LM-OTS public keys with a Merkle tree of height h. Each leaf of the tree is the hash of an LM-OTS public key, while each internal node is computed as the hash of its two children. The root node serves as the LMS public key.
The procedure is as follows:

1. For each leaf index $q \in \{0, \ldots, 2^h - 1\}$:
   (a) Generate the LM-OTS public key $\mathsf{pk}_{\mathrm{OTS}}^{(q)}$ corresponding to the private key $\mathsf{sk}_{\mathrm{OTS}}^{(q)}$.
   (b) Compute the leaf node:

$$\mathsf{Leaf}_q = H(I \parallel u32(q) \parallel u16(D_{\mathsf{LEAF}}) \parallel \mathsf{pk}_{\mathrm{OTS}}^{(q)}),$$

   where D_{LEAF} is a fixed domain-separation constant.

2. For each internal node at level j of the tree:

$$\mathsf{Node}_{j,k} = H(I \parallel u32(j) \parallel u32(k) \parallel u16(D_{\mathsf{INTR}}) \parallel \mathsf{Node}_{j+1,2k} \parallel \mathsf{Node}_{j+1,2k+1}),$$

   where j is the level index and k is the node index at that level.

3. The root of the tree is obtained at level 0, index 0:

$$\mathsf{Root} = \mathsf{Node}_{0,0}.$$

The LMS public key is then defined as:

$$\mathsf{pk}_{\mathrm{LMS}} = (\texttt{LMS\_type}, \texttt{LM-OTS\_type}, I, \mathsf{Root}),$$

where `LMS_type` encodes the chosen tree height h and hash function.
Intuitively, the Merkle tree compresses the authentication of 2^h LM-OTS public keys into a single root digest. Any signature has to carry an authentication path of intermediate nodes, allowing the verifier to recompute the root and confirm that the used LM-OTS key belongs to the tree.

3.4.4 Signature Generation

An LMS signature on a message M combines two components:

- An LM-OTS signature σ_{OTS} generated using the LM-OTS key pair at leaf q.

- An *authentication path* consisting of the h sibling nodes that allow the verifier to recompute the Merkle root from the chosen leaf.

The process is as follows:

1. Select the next unused LM-OTS private key $\mathsf{sk}_{\mathrm{OTS}}^{(q)}$ at leaf index q.
2. Generate the LM-OTS signature of the message:
$$\sigma_{\mathrm{OTS}} \leftarrow \mathsf{LM\text{-}OTS.Sign}(M, \mathsf{sk}_{\mathrm{OTS}}^{(q)}).$$
3. Collect the authentication path $\mathcal{A} = (\mathsf{Auth}_0, \ldots, \mathsf{Auth}_{h-1})$, where Auth_j is the sibling node of the node on the path from leaf q to the root at level j.
4. Output the LMS signature:
$$\sigma_{\mathrm{LMS}} = (q, \sigma_{\mathrm{OTS}}, \mathcal{A}).$$

Intuitively, the LM-OTS signature authenticates the message, while the authentication path demonstrates that the LM-OTS public key used belongs to the Merkle tree whose root is the LMS public key. Together, these components allow any verifier to confirm that the message was signed by the LMS key pair.

3.4.5 Signature Verification

Given a message M, an LMS public key $\mathsf{pk}_{\mathrm{LMS}} = (I, T[1])$, and a signature $\sigma_{\mathrm{LMS}} = (q, \sigma_{\mathrm{OTS}}, \mathcal{A})$, the verifier proceeds as follows:

1. Parse the leaf index q, the LM-OTS signature σ_{OTS}, and the authentication path $\mathcal{A} = (\mathsf{Auth}_0, \ldots, \mathsf{Auth}_{h-1})$.
2. Recover the LM-OTS public key candidate by verifying the OTS signature:
$$K_c \leftarrow \mathsf{LM\text{-}OTS.Verify}(M, \sigma_{\mathrm{OTS}}, I, q).$$
If the LM-OTS verification fails, reject the signature.
3. Compute the leaf node:
$$N^{(0)} = H(I \parallel u32str(q) \parallel u16str(D_{\mathrm{LEAF}}) \parallel K_c).$$
4. Iteratively compute the parent nodes up to the root using the authentication path:
$$N^{(j+1)} = H(I \parallel u32str(\lfloor q/2^{j+1} \rfloor) \parallel u16str(D_{\mathrm{INTR}}) \parallel (N^{(j)} \parallel \mathsf{Auth}_j)),$$
where D_{INTR} is the domain-separation constant for internal nodes, and the concatenation order of $(N^{(j)}, \mathsf{Auth}_j)$ depends on whether $N^{(j)}$ is a left or right child.

5. After h iterations, obtain the candidate root $N^{(h)}$.

6. Accept the signature if and only if:

$$N^{(h)} = T[1],$$

   i.e., the recomputed root matches the LMS public key.

Essentially, the verifier checks that the LM-OTS signature is valid and that the corresponding LM-OTS public key is correctly associated with the Merkle tree defined by the LMS public key.

3.4.6 Recommended Parameter Sets

While the LMS and LM-OTS algorithms can be instantiated with a wide range of parameters, NIST has profiled a subset of secure and efficient options in Special Publication SP 800-208 [12]. These parameter sets are intended for deployment in practical systems and balance security, signature size, and performance.
In particular, SP 800-208 recommends the following families:

- LM-OTS with $n = 32$ bytes (based on SHA-256), and Winternitz parameters $w \in \{1,2,4,8\}$.
- LMS with $m = 32$ bytes (hash length) and tree height $h \in \{5,10,15,20,25\}$.

Each LMS instance must specify both an LMS parameter set and a compatible LM-OTS parameter set. Table 3.2 summarizes the approved combinations.

LMS Parameter	**Hash Function**	**Height** h	**LM-OTS Type**	**Hash Output** n
LMS_SHA256_M32_H5	SHA-256	5	LMOTS_SHA256_N32_W1/2/4/8	32
LMS_SHA256_M32_H10	SHA-256	10	LMOTS_SHA256_N32_W1/2/4/8	32
LMS_SHA256_M32_H15	SHA-256	15	LMOTS_SHA256_N32_W1/2/4/8	32
LMS_SHA256_M32_H20	SHA-256	20	LMOTS_SHA256_N32_W1/2/4/8	32
LMS_SHA256_M32_H25	SHA-256	25	LMOTS_SHA256_N32_W1/2/4/8	32

Table 3.2: NIST-approved LMS and LM-OTS parameter sets [12]

These parameter sets provide flexibility across applications: smaller tree heights (e.g., $h = 5$) are appropriate for lightweight use cases where only a limited number of signatures are required, while larger heights (e.g., $h = 25$) enable millions of signatures at the cost of larger verification paths and higher computation.

3.5 Hash Complexity of LM-OTS and LMS

The efficiency of LMS is dominated by the number of invocations of the underlying hash function. By analyzing key generation, signature generation, and verification, it becomes possible to estimate the computational burden for different parameter sets, which in turn motivates the use of hardware acceleration.

Key Generation

The LMS key generation procedure requires computing the public key of every LM-OTS leaf and then building the Merkle tree. For each of the 2^h leaves, the LM-OTS public key generation involves $p(2^w - 1)$ hash function evaluations (to complete the Winternitz chains) plus one additional hash to compress them into the OTS public key K. The 2^h OTS public keys are then hashed as leaves of the Merkle tree, and an additional $2^h - 1$ hashes are needed to compute the internal nodes.
Hence, the total number of hash evaluations for LMS key generation is:

$$\#H_{\text{keygen}} = 2^h\Big(p(2^w - 1) + 1\Big) + (2^{h+1} - 1).$$

Signature Generation

For each message, the LM-OTS signature is obtained by partially iterating each Winternitz chain according to the w-bit digits of the message digest and its checksum. On average, this requires half of the chain length:

$$\#H_{\text{OTS, sign}} \approx \frac{p(2^w - 1)}{2}.$$

An additional hash is required for the message digest Q, while the Merkle authentication path does not require new hashes if precomputed. If authentication paths are computed on the fly, an additional traversal cost of $O(h)$ hashes must be considered.

Signature Verification

Verification requires completing the remaining part of each Winternitz chain, i.e.,

$$\#H_{\text{OTS, verify}} \approx \frac{p(2^w - 1)}{2}.$$

In addition, one hash is needed to recompute Q, one to compress the chain outputs into the OTS public key candidate K_c, one for the leaf hash, and exactly h hashes for recomputing the authentication path.

Thus, the total verification cost is:

$$\#H_{\text{verify}} \approx \frac{p(2^w - 1)}{2} + (h + 3).$$

Table 3.3 summarizes the resulting complexity for different combinations of w and h and highlights the trade-off between signature size and computational effort: increasing w reduces the signature length but significantly increases the number of hashes required.

w	p	h	KeyGen Hashes (total)	Sign OTS Hashes (avg)	Verify Total Hashes (avg)
1	265	5	8,575	132	140
1	265	10	274,431	132	145
1	265	15	8,781,823	132	150
1	265	20	281,018,367	132	155
1	265	25	8,992,587,775	132	160
2	133	5	12,863	199	207
2	133	10	411,647	199	212
2	133	15	13,172,735	199	217
2	133	20	421,527,551	199	222
2	133	25	13,488,881,663	199	227
4	67	5	32,255	502	510
4	67	10	1,032,191	502	515
4	67	15	33,030,143	502	520
4	67	20	1,056,964,607	502	525
4	67	25	33,822,867,455	502	530
8	34	5	277,535	4,335	4,343
8	34	10	8,881,151	4,335	4,348
8	34	15	284,196,863	4,335	4,353
8	34	20	9,094,299,647	4,335	4,358
8	34	25	291,017,588,735	4,335	4,363

Table 3.3: Hash complexity of LMS as a function of w and h (SHA-256, $n = 32$).

As shown, key generation is by far the most expensive step, scaling exponentially with the tree height h. While this cost is paid only once per key pair, it highlights the need for hardware acceleration in constrained environments. The dominant cost for signing and verification arises from the LM-OTS Winternitz chains, which grow rapidly with w.
To better illustrate the trends, Figures 3.1 and 3.2 present graphical comparisons.

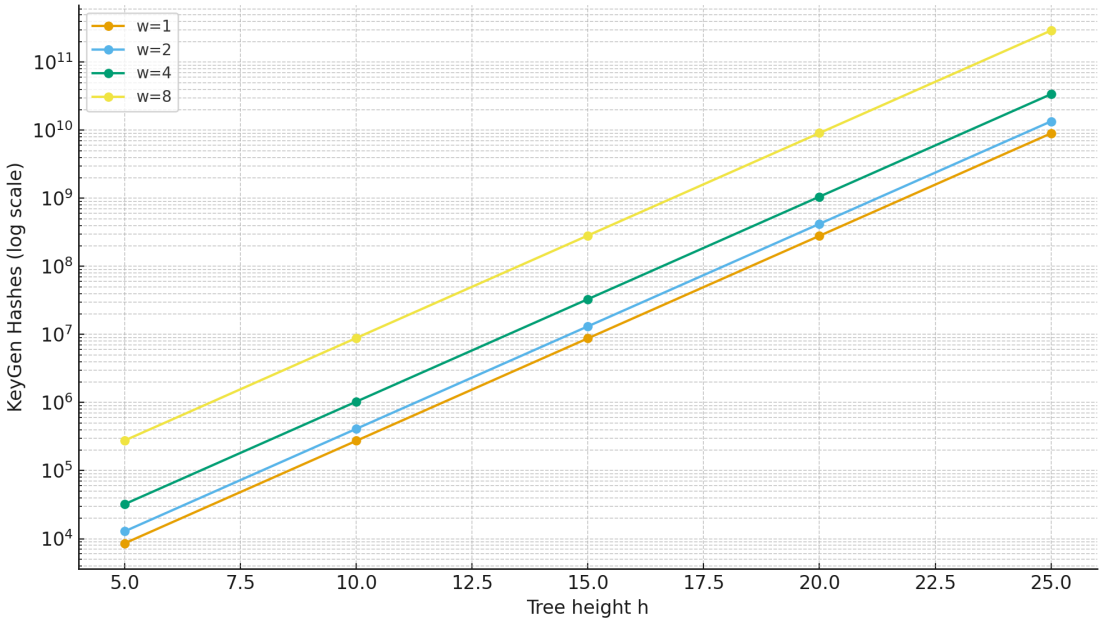


Figure 3.1: Key generation cost vs. tree height h (log scale). Each curve corresponds to a Winternitz parameter w.

As expected, the number of hash evaluations required for key generation grows exponentially with the tree height h, due to the 2^h leaves in the Merkle tree. The choice of w strongly influences the slope, since the LM-OTS key generation dominates the computation. While this cost is amortized over the lifetime of the key pair, acceleration might be necessary to keep key generation feasible.

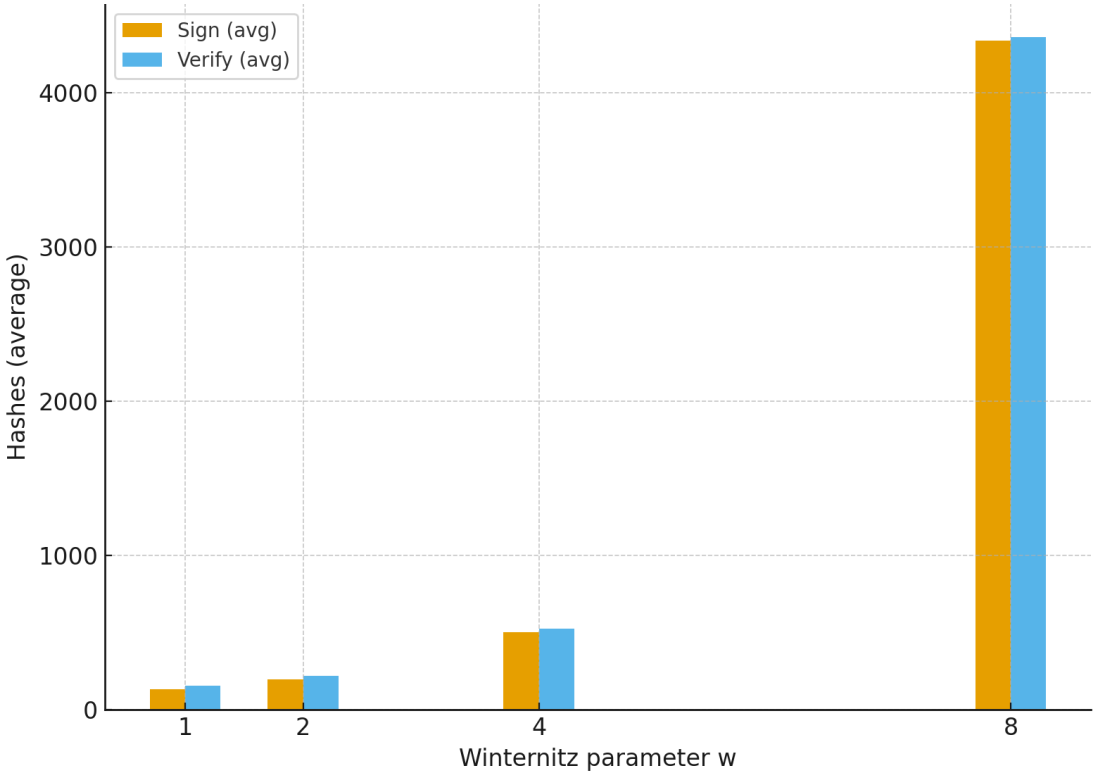


Figure 3.2: Average signing and verification costs vs. Winternitz parameter w for fixed tree height $h = 20$.

Figure 3.2 emphasizes the trade-off between signature size and performance. Increasing w reduces the signature length, but the signing and verification costs grow rapidly from roughly 130 hashes at $w = 1$ to over 4,000 hashes at $w = 8$. Verification requires only a small additional overhead of $h + 3$ hashes for the authentication path. These trends justify the need to carefully select LMS parameters depending on the deployment scenario, balancing compactness and computational feasibility.

3.6 Hierarchical Signatures (HSS)

In scenarios where reducing the public key generation time is critical, the Hierarchical Signature System (HSS) can be used as an extension of LMS. HSS builds a hierarchy of L LMS trees, where the public key of the first tree (level 0) becomes the HSS public key, and each LMS private key at level i signs the public key of the tree at level $i + 1$. The last LMS private key in the hierarchy (level $L - 1$) is then used to sign the actual messages.
The HSS public key is the root of the level 0 LMS tree, and an HSS signature

contains the chain of LMS signatures along the path from the root tree down to the message. Verification is performed by recursively validating each LMS signature in this chain.
The main advantage of HSS compared to a single, very tall LMS tree is efficiency: only the top-level LMS tree must be generated in order to publish the HSS public key. The lower-level LMS trees can be generated on demand as signatures are needed. This design significantly reduces the cost of public key generation, while still allowing a very large total number of signatures. For instance, an HSS with $L = 3$ levels of height $h = 10$ each can support 2^{30} signatures in total, while requiring only 2^{10} one-time keys to be computed initially. In contrast, a flat LMS tree with 2^{30} leaves would require generating all one billion leaves in advance.
Another benefit of HSS is flexibility: different levels of the hierarchy can use different LMS parameter sets, allowing implementers to balance performance and storage needs. However, the parameter set at a given level must remain fixed throughout its lifetime, in order to ensure interoperability and consistent verification.

3.7 Key Lifetime Under Stateful Signing

Because LMS is stateful, a key pair can produce at most 2^h signatures (one per leaf). This property has direct implications on the operational lifetime of a key. If signatures are generated at a constant rate r, then the maximum lifetime of a key pair is given by:

$$T_{\text{life}} \;=\; \frac{2^h}{r} \text{ seconds.}$$

For other units:

$$T_{\text{days}} = \frac{2^h}{r \cdot 86{,}400}, \qquad T_{\text{years}} \approx \frac{2^h}{r \cdot 86{,}400 \cdot 365.25}.$$

In an HSS stack with d LMS layers of heights $h_1, \ldots, h_d$, the total signature capacity multiplies:

$$N_{\text{sign}} \;=\; \prod_{i=1}^{d} 2^{h_i} \;=\; 2^{\sum_i h_i}, \qquad T_{\text{life}}^{(\text{HSS})} \;=\; \frac{2^{\sum_i h_i}}{r}.$$

Table 3.4 shows illustrative examples of the key lifetime for different tree heights h, assuming various signing rates.

h	**Max signatures** 2^h	$r = 1/\mathbf{s}$	$r = 100/\mathbf{day}$	$r = 10/\mathbf{day}$	$r = 1/\mathbf{day}$
5	32	32 s ($\approx$ 0.5 min)	$\approx$ 8 hours	$\approx$ 3 days	$\approx$ 1 month
10	1,024	$\approx$ 17 min	$\approx$ 12 days	$\approx$ 4 months	$\approx$ 3 years
15	32,768	$\approx$ 9 hours	$\approx$ 11 months	$\approx$ 9 years	$\approx$ 89 years
20	1,048,576	$\approx$ 12 days	$\approx$ 28 years	$\approx$ 287 years	$\approx$ 2,872 years
25	33,554,432	$\approx$ 1 year	$\approx$ 920 years	$\approx$ 9,200 years	$\approx$ 92,000 years

Table 3.4: Illustrative LMS key lifetimes for different tree heights h and signing rates r.

The appropriate choice of h depends on the expected total number of signatures over the key's lifetime. In scenarios with few signatures per year like long-lived embedded/automotive deployments, even small heights can be sufficient: h=5 returns 32 signatures, and h=10 returns 1,024, which already covers many years at low signing rates. Larger values (e.g., h=15 or h=20) provide additional safety margin at the cost of longer authentication paths and higher key-generation effort. To better illustrate the practical implications of the limited number of signatures in LMS, Figure 3.3 reports the effective key lifetime under different signing rates and tree heights.

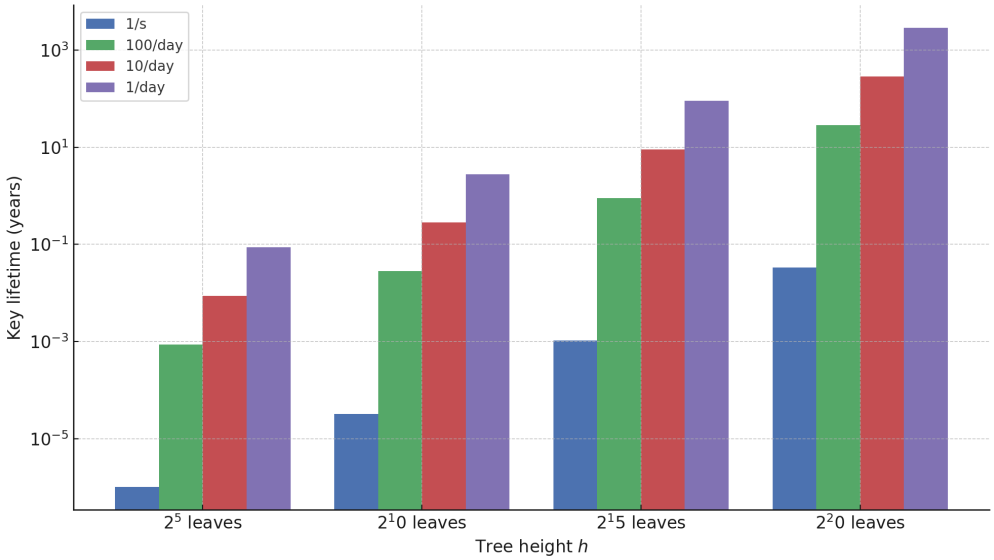


Figure 3.3: LMS key lifetime (in years) as a function of tree height h and signing rate r.

Chapter 4

Hardware–Software Co-Design for LMS

4.1 SHA-256 Hardware Accelerator Porting

The hardware acceleration of SHA-256 is the foundation of the co-design approach adopted in this thesis. The accelerator is based on an open-source design developed for the Zybo board [13], which was ported and adapted to the PYNQ-Z2. Although the two boards share the Xilinx Zynq-7000 SoC family, differences in peripherals and integration required a complete re-synthesis and repackaging of the IP.
The accelerator implements the SHA-256 compression function in hardware, organized around a pipelined datapath and a control unit. The datapath is responsible for processing 32-bit input words and updating the intermediate state of the hash, while the control unit orchestrates the sequencing of rounds via a finite state machine. Optimizations such as operation rescheduling and carry-save addition reduce the critical path, enabling higher operating frequencies with limited area overhead, this design therefore trades an increased number of cycles per block for improved synthesis timing and smaller resource usage.
Communication between the processing system and the accelerator is realized through an AXI4-Lite interface, exposing a set of memory-mapped registers. These registers allow the processor to send data words, control the computation, and retrieve the final hash values. In particular, the `status_reg` plays a central role in coordinating the execution, as it contains the control and handshake signals used by both the hardware and the processor to synchronize operations. Figure 4.1 illustrates the memory map of the accelerator. The data register is used to write input words, while the eight hash registers (`h0`–`h7`) hold the 256-bit result once a message block has been processed. The `status_reg` contains four control bits: `msg_last` indicates that the current word is the last of the message, `hash_ack`

acknowledges that the computed hash has been read, `hash_valid` signals that the hash is ready, and `msg_ready` indicates that the accelerator is ready to receive a new word.

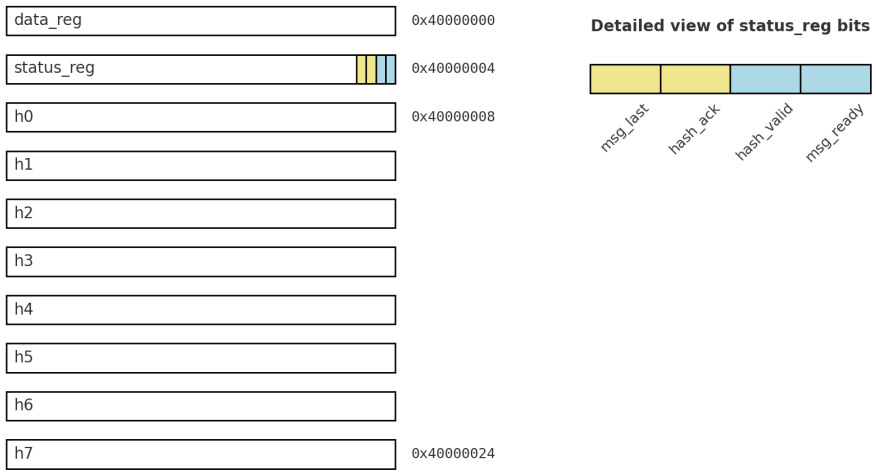


Figure 4.1: Memory map of the SHA-256 accelerator. The `status_reg` contains control bits: `msg_last`, `hash_ack`, `hash_valid`, and `msg_ready`.

The HDL sources were imported into Vivado 2022.1, synthesized for the Zynq-7020 device on the PYNQ-Z2, and packaged as a reusable IP. The block was then integrated into a Zynq block design and connected to the processing system via AXI interconnect. Vivado reports indicate resource utilization of less than 10% of available LUTs and registers. After validation, the final design was implemented and the FPGA bitstream generated.

On the software side, the accelerator was initially controlled through a device driver providing read and write access to the AXI-mapped registers. The driver was compiled with the cross-compiler provided in Vitis 2022.1 (`arm-xilinx-linux-gnueabi-gcc`) and deployed in the PYNQ Linux distribution. Two testbenches were developed to verify correct operation: a Python-based testbench using PYNQ overlays, which enabled quick functional checks, and a C-based testbench using the installed driver, which validated low-level operation and served as the basis for benchmarking code.

These steps ensured that the accelerator was correctly ported and integrated on the PYNQ-Z2, providing a functional hardware building block to be later combined with the LMS software implementation.

4.2 Device Driver Development

The SHA-256 hardware accelerator is accompanied by a Linux device driver, provided as part of the original open-source project [13]. This driver served as the foundation for integration on the PYNQ-Z2, as it already supported the basic operations required to communicate with the accelerator through the AXI-mapped registers.
The driver implements the standard operations `open`, `release`, `read`, and `write`. In its original form, the `read` operation retrieves data from the accelerator registers (e.g., the `status_reg` or the hash words stored in `h0`–`h7`), while `write` allows the processor to send new input words or control values. At the kernel level, these calls rely on `ioread32()` and `iowrite32()`, which provide direct access to the mapped addresses of the accelerator.
The driver was compiled against the Xilinx Linux kernel sources, cloned from the official repository [14], using a standard out-of-tree build process.
In practice, the compilation required setting the cross-compiler provided by Vitis 2022.1, preparing the kernel headers, and then building the module as follows:

```
$ export CROSS_COMPILE=/home/francesco/Vitis/2022.1/gnu/aarch32/
    lin/gcc-arm-linux-gnueabi/x86_64-petalinux-linux/usr/bin/arm-
    xilinx-linux-gnueabi/arm-xilinx-linux-gnueabi-

$ make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE modules_prepare
$ make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE \
       -C /home/francesco/Desktop/Tesi/DRIVER/kernel-headers \
       M=$(pwd) modules
```

Listing 4.1: Cross-compilation of the driver

Once compiled, the module could be loaded on the PYNQ-Z2 board:

```
$ insmod sha256_driver.ko
```

Listing 4.2: Loading the driver

Functionally, the original driver creates a character device entry at `/dev/sha256`, exposing the accelerator registers to user-space applications. When opened, the device accepts standard `read` and `write` calls:

- **`read`** retrieves a 32-bit word from the accelerator using `ioread32()`, which is typically employed to poll the `status_reg` or to read the final hash values stored in registers `h0`–`h7`.
- **`write`** accepts a 32-bit word from user space via `copy_from_user()` and sends it to the accelerator with `iowrite32()`, enabling the processor to provide input words and control signals.

The `open` and `release` functions are simple placeholders and do not perform additional logic. The driver's probe function handles resource allocation by reserving the physical memory range of the accelerator and remapping it into the kernel's virtual address space with `ioremap()`, while the remove function cleans up these resources upon module unload.
This design provides a correct and minimal interface between software and hardware. However, because each 32-bit access involves a system call, the overhead becomes significant when hashing larger messages.

Optimized Driver

While the original driver provided a correct interface to the accelerator, its performance was limited by the overhead of repeated `read` and `write` system calls. Each 32-bit access required a user–kernel context switch, which became a significant bottleneck when processing messages of non-trivial length.
The most significant addition is the implementation of the `mmap()` operation, shown in Listing 4.3. This function maps the accelerator's physical AXI address space into the calling process's virtual memory. By marking the memory region as non-cached and using `remap_pfn_range()`, the mapping ensures that all register accesses in user space are consistent and coherent with the underlying hardware.

```
1  static int sha256_mmap(struct file *file, struct vm_area_struct *
       vma) {
2      unsigned long phys = BASE_ADDR;
3      unsigned long vsize = vma->vm_end - vma->vm_start;
4
5      if (vsize > MEM_SIZE)
6          return -EINVAL;
7
8      // Mark memory as non-cached to ensure consistency
9      vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
10
11     // Map physical address into user space
12     return remap_pfn_range(vma,
13                            vma->vm_start,
14                            phys >> PAGE_SHIFT,
15                            vsize,
16                            vma->vm_page_prot);
17 }
```

Listing 4.3: `mmap()` implementation in the optimized driver

Through this mechanism, the driver enables direct memory-mapped access to the accelerator's registers, eliminating the overhead of system calls for each word transfer. Applications can simply obtain a pointer to the mapped region and issue loads and stores directly, yielding a significant reduction in latency.

The resulting driver preserves compatibility with the character device interface (`/dev/sha256`) while adding an `mmap()` file operation.
The structure of the optimized driver is therefore:

- **Initialization and cleanup**: reserve the AXI memory region, remap it into kernel space, and register the character device under `/dev/sha256`.
- **`read`/`write`**: still available for compatibility, performing 32-bit transfers with `ioread32()` and `iowrite32()`.
- **`mmap`**: new operation that maps the hardware registers directly into user space, enabling high-performance access without repeated system calls.

With this modification, benchmark applications can use `mmap()` to establish a persistent mapping to the accelerator's register space, drastically reducing per-access latency. This proved to be the most effective optimization, and the impact of this change is quantified in Chapter 5.
The full source code of the optimized driver is included in Appendix A.1.

Python Overlay Testbench

Before developing a custom C application, the accelerator was first tested through the PYNQ overlay framework. Overlays are a central feature of the PYNQ platform: they are FPGA bitstreams that can be dynamically loaded into the programmable logic at runtime, together with a Python object model that exposes the hardware IP cores as software-accessible objects. This makes it possible to rapidly prototype and interact with hardware directly from Python, without rebuilding or rebooting the system.
In this case, the synthesized bitstream of the accelerator (`sha256_wrapper.bit`) was loaded as an overlay, which made the SHA-256 IP core accessible under the identifier `overlay.sha256_ctrl_axi_0`. The Python testbench, shown in Appendix A.2, was designed to provide end-to-end functional testing of the accelerator. The testbench implemented the following steps:

1. **Message padding**: a series of helper functions implemented the standard SHA-256 padding procedure (appending a '1' bit, adding zeroes until the length modulo 512 equals 448, and appending the 64-bit message length).
2. **Word transfer**: the padded message was divided into 32-bit words and sequentially written to the accelerator through its AXI registers. The `msg_last` bit in the `status_reg` was set for the final word.
3. **Hash retrieval**: the program polled the `hash_valid` flag until the computation was complete, then read the eight 32-bit words from registers `h0`–`h7`.

4. **Result output**: the 256-bit digest was printed in hexadecimal format and compared against known test vectors.

The testbench provided an interactive loop in which the user could type arbitrary messages, which were then padded, sent to the accelerator, and hashed in real time. An excerpt of the core hashing function is shown in Listing 4.4.

```
def hash_message(padded_binary):
    num_words = len(padded_binary) // 32
    for i in range(num_words):
        word = padded_binary[i*32:(i+1)*32]
        is_last = (i == num_words - 1)
        write_word(word, is_last)

    while sha256_accel.read(0x4) & 0x2 == 0:  # wait for
   hash_valid
        pass

    hash_words = []
    for offset in range(0x8, 0x28, 4):  # h0 to h7
        h = sha256_accel.read(offset)
        hash_words.append(h)

    sha256_accel.write(0x4, 0x4)  # set hash_ack
    return hash_words
```

Listing 4.4: Hashing procedure in the Python overlay testbench

This Python-based approach allowed rapid functional validation of the accelerator immediately after synthesis. Although not optimized for speed, it confirmed that the hardware logic was functionally correct and provided a convenient baseline before moving to the lower-level C testbench.

C Driver Testbench

After the functional verification performed with the Python overlay, the accelerator was also tested through a low-level C testbench running on the PYNQ-Z2. Unlike the overlay approach, this method relied directly on the installed Linux driver, interacting with the device through the character interface `/dev/sha256`. In this way, the accelerator was validated in its final deployment environment, using the same stack that would later support benchmarking.
The testbench was split into a header file, `sha256_acc_tb.h`, and two source files, `sha256_acc_tb.c` and `sha256_testbench.c`, which implemented the actual testing routines. Once compiled and executed on the board, the application guided the complete hashing process: opening the character device, padding the input message, transferring each 32-bit word to the accelerator, setting the `msg_last` flag

for the final word, and finally polling the `hash_valid` bit until the digest became available.
The eight 32-bit words representing the final hash were then read from the accelerator and printed in hexadecimal format, to be compared against a reference software implementation. An excerpt of the code responsible for writing words to the accelerator is reported in Listing 4.5.

```
1 word_to_send = ((uint32_t*)&num_bits)[1];
2 if (pwrite(fd, &word_to_send, WORD_BYTES, DATA_IN_ADDR) < 4) {
3     perror("failed to write word");
4 }
```

Listing 4.5: Writing input words to the accelerator in the C testbench

This testbench confirmed both correctness and stability of the accelerator when accessed from user space, and at the same time it provided the foundation for the performance evaluation reported in Chapter 5.

4.3 Software Implementation of LMS

The software side of this work builds upon the reference LMS and LM-OTS implementation provided in the IETF RFC [1] and maintained by Cisco as the `hash-sigs` project [15]. This implementation was chosen because, at the time of writing, OpenSSL [16] does not yet provide support for LMS key generation or signature creation; support is planned to appear only for verification in version 3.6. The original codebase is organized around a set of core modules and a demonstration program (`demo.c`), which together implement the essential operations of the scheme: key generation, signature generation, and signature verification. The demo program exposes these capabilities through a command-line interface, allowing the user to select parameter sets, generate keys, sign arbitrary input files, and verify signatures against stored public keys.
In its unmodified form, the demo application is intended to be portable and efficient, and can execute on multicore systems. For the purposes of this thesis, however, the implementation was adapted to better reflect the characteristics of embedded systems and to ensure reproducibility in benchmarking. Specifically, the main function of `demo.c` was modified to bind execution explicitly to CPU 0, enforcing a single-core execution model. This change guarantees that all measured performance results correspond to a deterministic single-threaded execution, eliminating variability due to the operating system's scheduler. No changes were made to the functionality of the program, which continues to support key generation, signing, and verification. The modification is shown in Listing 4.6.

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(0, &set); // Use core 0

if (sched_setaffinity(0, sizeof(set), &set) == -1) {
    perror("sched_setaffinity");
    return 1;
}

printf("Running on core 0\n");
```

Listing 4.6: Modification in `demo.c` to enforce single-core execution

The file `hash.c` implements the internal hashing interface used throughout the LMS and LM-OTS codebase. In the original RFC implementation [15], all hashing operations were performed using the OpenSSL software implementation of SHA-256, both for single-shot hashes and for incremental hashing where the message is processed in multiple chunks. This design provided flexibility but did not exploit hardware acceleration.
To integrate the accelerator, the file was modified by introducing a conditional compilation flag (`USE_HW_ACCEL_SHA256`). When this flag is enabled, single-shot SHA-256 calls are redirected to the hardware accelerator, while retaining the original OpenSSL-based path for software execution. The modification can be seen in Listing 4.7.

```
#if USE_HW_ACCEL_SHA256
    hw_hash(message, message_len, result);  // hardware path
#else
    SHA256_Init(&ctx->sha256);
    SHA256_Update(&ctx->sha256, message, message_len);
    SHA256_Final(result, &ctx->sha256);     // software path
#endif
```

Listing 4.7: Hardware acceleration hook in `hash.c`

This approach allows the same LMS codebase to operate seamlessly in either mode: pure software, or hardware-assisted. In practice, the hardware path invokes the routines defined in `sha256_acc.c/h`, which handle the communication with the accelerator through its driver. For completeness, the full source code of this interface is reported in Appendix A.4. A crucial optimization introduced in these routines is the use of `mmap()` to directly map the accelerator's AXI registers into user space. As shown in Listing 4.8, the file descriptor for the device is opened once, and the register space is mapped into virtual memory. From this point onward, all accesses to the accelerator are performed via the `REG(offset)` macro, which compiles down to a simple memory load or store, eliminating the overhead of repeated system calls.

```c
#define REG(offset) (*(volatile uint32_t *)((uint8_t *)sha256_regs
    + (offset)))

volatile uint32_t *sha256_regs = NULL;

void sha256_acc_init() {
    sha256_fd = open(DEVICE_FILE, O_RDWR | O_SYNC);
    if (sha256_fd < 0) { perror("open"); exit(1); }

    sha256_regs = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
                       MAP_SHARED, sha256_fd, 0);
    if (sha256_regs == MAP_FAILED) { perror("mmap"); exit(1); }
}
```

Listing 4.8: Memory-mapping of the accelerator registers in `sha256_acc.c`

In addition, the file `hss_pthread.c`, which normally handles multithreading support in the reference implementation, was modified to disable threading entirely. The initialization function `hss_thread_init` was adapted to immediately return 0, as shown in Listing 4.9, thereby forcing single-thread execution across the LMS codebase and ensuring consistency in the benchmarking environment.

```c
/* Allocate a thread control structure */
struct thread_collection *hss_thread_init(int num_thread) {
    return 0; // Force single thread execution for benchmarking
}
```

Listing 4.9: Modification in `hss_pthread.c` to disable multithreading

Incremental Hashing and its Limitation.

An important detail is that the hardware accelerator only supports complete SHA-256 computations over a fully padded message block. It does not maintain an internal state across multiple calls, therefore, functions such as `hss_init_hash_context`, `hss_update_hash_context`, and `hss_finalize_hash_context` remain bound to the software implementation, as shown in Listing 4.10.

```c
void hss_update_hash_context(int h, union hash_context *ctx,
                             const void *msg, size_t len_msg) {
    switch (h) {
    case HASH_SHA256:
        SHA256_Update(&ctx->sha256, msg, len_msg); // software
    only
        break;
    }
}
```

Listing 4.10: Incremental hashing remains software-only

This distinction is important: many LMS operations, particularly when hashing long messages or when building authentication paths, rely on incremental hashing. Only those calls that perform a single complete hash (e.g., hashing seeds, nonces, or intermediate nodes) benefit from hardware acceleration. As a result, the accelerator cannot replace all SHA-256 invocations in the system, and the benchmarking results in Chapter 5 reflect this partial acceleration.
In summary, the modifications to `hash.c` enable the use of hardware acceleration only for single-shot hashes, while incremental hashing remains bound to the software path. This outcome is not a design choice but a limitation of the current accelerator, which cannot preserve state across multiple calls. A more advanced accelerator would be required to fully replace all SHA-256 invocations within LMS.

4.4 Summary and Outlook

In this chapter, the SHA-256 hardware accelerator was successfully ported to the PYNQ-Z2, integrated through a Linux device driver, and connected to the reference LMS implementation. Several modifications were introduced to ensure reproducibility and to enable hardware offloading within LMS: the driver was optimized using memory-mapped I/O, the LMS software was constrained to single-core, single-thread execution, and the hashing interface was extended to selectively invoke the accelerator for single-shot SHA-256 computations.
At the same time, some limitations were highlighted. In particular, incremental hashing remains bound to the software path, since the accelerator cannot preserve state across multiple calls. As a result, only a subset of the SHA-256 invocations within LMS benefit from acceleration, while those relying on incremental hashing remain executed in software. A detailed quantification of the proportion of accelerated versus non-accelerated hashes is presented in Chapter 5, where this limitation is reflected in the performance results.
With the software and hardware components integrated and validated, the next chapter introduces the benchmarking methodology and presents a detailed comparison between the pure software and hardware-assisted implementations of LMS.

Chapter 5

Benchmarking and Performance Evaluation

5.1 Reproducibility and Setup

The benchmarking environment was designed to be fully reproducible and automated. At the top level of the project directory, a shell script (`benchmark.sh`) orchestrates the execution of all experiments. Depending on whether the software-only or the hardware-accelerated implementation is under test, the script is run in the `lms_software_benchmark` or `lms_accelerated_benchmark` directory.
Each of these benchmark directories mirrors the structure of the original `hash-sigs` repository, with the accelerated version including in addition the files required to interface with the SHA-256 hardware accelerator. A placeholder binary file (`firmware.bin`) is also included in the top folder to simulate the firmware image that is signed and verified during the LMS operations.
Within each benchmark directory, an `outputs/` folder organizes the results by Winternitz parameter. Specifically, four subdirectories (`W_1`, `W_2`, `W_4`, `W_8`) are created, each containing a set of `LMS_SHA256_M32_H{5,10,15,20,25}_W{X}` subfolders. These in turn correspond to the different LMS parameter sets used in the experiments.
Inside each parameter set folder, a `Makefile` drives the benchmarking process, and three additional directories (`profile_genkey`, `profile_sign`, `profile_verify`) are maintained to store the results of the different LMS operations. The top-level `benchmark.sh` script iterates over these directories, invokes the Makefiles, and consolidates the outputs into log files.
For completeness, the source code of the benchmarking infrastructure is provided in Appendix A.5. This includes the `benchmark.sh` script, which automates the entire campaign, and a representative `Makefile` for the profiling of `genkey`, `sign`,

and `verify`. These listings document precisely how the experiments were executed and how the outputs were collected for analysis.

5.2 Benchmarking Methodology

The benchmarking strategy was layered, starting from simple execution time measurements and progressively adopting more advanced profiling tools as the analysis deepened. This ensured both broad coverage of the LMS operations and a detailed investigation of the observed bottlenecks.

5.2.1 C Library Timing

The first stage of measurement relied on the standard C library timing functions, specifically `clock_gettime()` from `time.h` [17]. Calls were inserted around the invocations of the key LMS functions: `keygen`, `sign`, and `verify`. This allowed end-to-end execution times for each operation to be obtained, expressed in wall-clock milliseconds. These results provide an intuitive baseline but are sensitive to system noise and do not reveal the internal distribution of computational costs.
A simplified example is shown in Listing 5.1, where a call to the `sign()` function is wrapped between two calls to `clock_gettime()` and the elapsed time is reported.

```
1  struct timespec start_sign, end_sign;
2  clock_gettime(CLOCK_MONOTONIC, &start_sign);
3
4  int ret = sign(argv[2], &argv[3]);
5
6  clock_gettime(CLOCK_MONOTONIC, &end_sign);
7  double elapsed_ms = (end_sign.tv_sec - start_sign.tv_sec) * 1000.0
       +
8                      (end_sign.tv_nsec - start_sign.tv_nsec) / 1e6;
9
10 printf("Sign time: %.3f ms\n", elapsed_ms);
```

Listing 5.1: Timing instrumentation using `clock_gettime()`

Equivalent wrappers were placed around `keygen()` and `verify()`, ensuring that all three LMS operations could be measured consistently.

5.2.2 Profiling with gprof

To understand how execution time was distributed inside the LMS implementation, the code was compiled with the `-pg` flag to enable profiling with `gprof` [18]. After each benchmark run, a `gmon.out` file was produced and converted into

a human-readable report (e.g., `gprof ${SRC}/demo gmon.out > profile_verify/profile_verify.txt`). This analysis highlighted which functions dominated the runtime during key generation, signing, and verification, thereby confirming the central role of SHA-256 in the computation.

5.2.3 Cycle-accurate Measurements with perf

Finally, to analyze the root causes of performance differences between the software and hardware-assisted implementations, the Linux `perf` tool was employed [19]. This allowed cycle-accurate statistics such as the number of executed instructions, cache misses, and branch mispredictions to be collected, while also being robust against variability due to operating system scheduling. `perf` was used particularly in the microbenchmarks that measured isolated SHA-256 operations, enabling a direct comparison between software-only and hardware-accelerated execution.

5.3 Software-only LMS Benchmarking

The first set of experiments was conducted using the unmodified software implementation of LMS. All parameter sets recommended in the NIST specification were evaluated, namely tree heights $h \in \{5,10,15,20,25\}$ combined with Winternitz parameters $w \in \{1,2,4,8\}$. For each configuration, the execution time of the three main operations (`keygen`, `sign`, and `verify`) was measured using `clock_gettime()`, and profiling data was collected with `gprof`.
To improve statistical confidence, each signing and verification measurement was repeated over 30 rounds. The raw results were post-processed with a custom Python script that extracted values from the logs, computed averages and standard deviations, and exported them in CSV format. The script is reported in Appendix A.6.
During the benchmarking, it was observed that all configurations with $h = 25$ exceeded practical runtimes for key generation on the PYNQ-Z2 board. Runs with $w = 1,2,4$ eventually completed but required many hours, while $w = 8$ was aborted due to the prohibitive runtime (estimated in the order of weeks). To handle this, the $h = 25, w = 8$ key pair was generated once on the Politecnico di Torino HPC cluster and transferred to the PYNQ-Z2, so that signing and verification could still be attempted for completeness.
However, an additional limitation was encountered: for all parameter sets with $h = 25$, the signing operation systematically failed on the PYNQ-Z2. The library reported an `hss_error_out_of_memory`, which in the reference implementation corresponds to a "`A malloc failure caused us to fail`" condition. This indicates that the embedded platform cannot allocate sufficient memory for the internal structures required at this parameter size. Consequently, only key generation times

could be measured for $h = 25$, while signing and verification benchmarks are limited to $h = 5$,10,15,20.

5.3.1 Key Generation Times

Key generation proved to be by far the most resource-intensive operation. Table 5.1 reports the average execution time of key generation on the PYNQ-Z2 for all parameter sets. The exponential dependency on tree height h is immediately visible: moving from $h = 20$ to $h = 25$ increases the runtime by roughly two orders of magnitude. At $w = 8$, the experiment was aborted after several days without completion.

h	$w = 1$	$w = 2$	$w = 4$	$w = 8$
5	38.625	39.750	52.563	338.200
10	1035.991	860.013	1419.075	10557.765
15	32678.435	26969.896	44733.633	337161.818
20	1047901.150	855383.434	1426530.702	10765737.466
25	33498680.063	27502367.549	45639979.067	timeout

Table 5.1: Software-only key generation times (ms) on PYNQ-Z2.

As discussed earlier, for $h = 25$ the generated keys were either aborted on the board (for $w = 8$) or completed after many hours (for $w = 1$,2,4). In practice, these results confirm that key generation dominates the computational cost of LMS at higher security parameters, and explains why such parameter sets are unsuitable for constrained platforms like the PYNQ-Z2.

Feasibility estimate via `openssl speed`

To give an intuitive sense of why LMS key generation becomes infeasible at large tree heights, the OpenSSL microbenchmark was executed:

```
1 $ openssl speed sha256
```

Listing 5.2: Measuring SHA-256 throughput on PYNQ

The tool runs repeated SHA-256 computations on 64-byte blocks for three seconds and reports the total operations. On the PYNQ, the output was approximately

$$523{,}076 \text{ hashes in } 3 \text{ s} \;\Rightarrow\; \texttt{hashes\_per\_sec} \approx 1.75 \times 10^5.$$

This benchmark is not restricted to single-core execution and is therefore not directly comparable to the controlled measurements presented earlier. However, it provides a useful order-of-magnitude reference when estimating runtimes.
For $w = 1,2,4$ the experiments on the PYNQ-Z2 completed successfully, although with very long runtimes in the order of hours. The measured key generation times were:

w	Measured time
1	33,498.7 s ($\sim$ 9.3 h)
2	27,502.4 s ($\sim$ 7.6 h)
4	45,640.0 s ($\sim$ 12.7 h)

Table 5.2: Measured key generation times for h=25 and $w = 1,2,4$ on the PYNQ-Z2.

The experiment for $w = 8$ was aborted due to the prohibitive runtime. Using the closed-form expression for LMS key generation

$$\#H_{\text{keygen}} = 2^h\Big(p(w)(2^w - 1) + 1\Big) + \Big(2^{h+1} - 1\Big),$$

together with the throughput estimate from `openssl speed`, the expected runtime is on the order of 19 days:

w	$p(w)$	$\#H_{\text{keygen}}$	Estimated time
8	34	$\approx 2.91 \times 10^{11}$	$\sim$ 19 d

Table 5.3: Estimated key generation time for h=25, w=8 on the PYNQ-Z2.

This estimate is consistent with the observation that completing the $h = 25, w = 8$ key generation on the PYNQ was impractical. For completeness, the key was generated once on the Politecnico di Torino HPC cluster and then transferred to the PYNQ-Z2 for subsequent signing and verification attempts, which however failed due to memory allocation errors.

5.3.2 Execution Times

Figures 5.1 and 5.2 report the averaged signing and verification latencies over 30 measurement rounds, with error bars indicating ± 1 standard deviation. Signing latencies increase with both h and w, while verification times remain remarkably stable across all parameters. The low variance in verification (standard deviation consistently below 1% relative to the mean) highlights the deterministic nature of

this operation: verification always requires traversing the full authentication path and performing a fixed number of hash computations. In contrast, signing exhibits higher dispersion, especially for $w = 8$. This behavior is a direct consequence of the Winternitz one-time signature scheme, where the number of hash iterations per signature depends on message-specific coefficients. For larger w, the possible chain lengths span a wider range (e.g., up to 255 hashes for $w = 8$), resulting in significant variation between different messages and, consequently, higher timing variance.
The complete set of numerical results, including averages and standard deviations for each (h, w) parameter set, is reported in Appendix B.1.

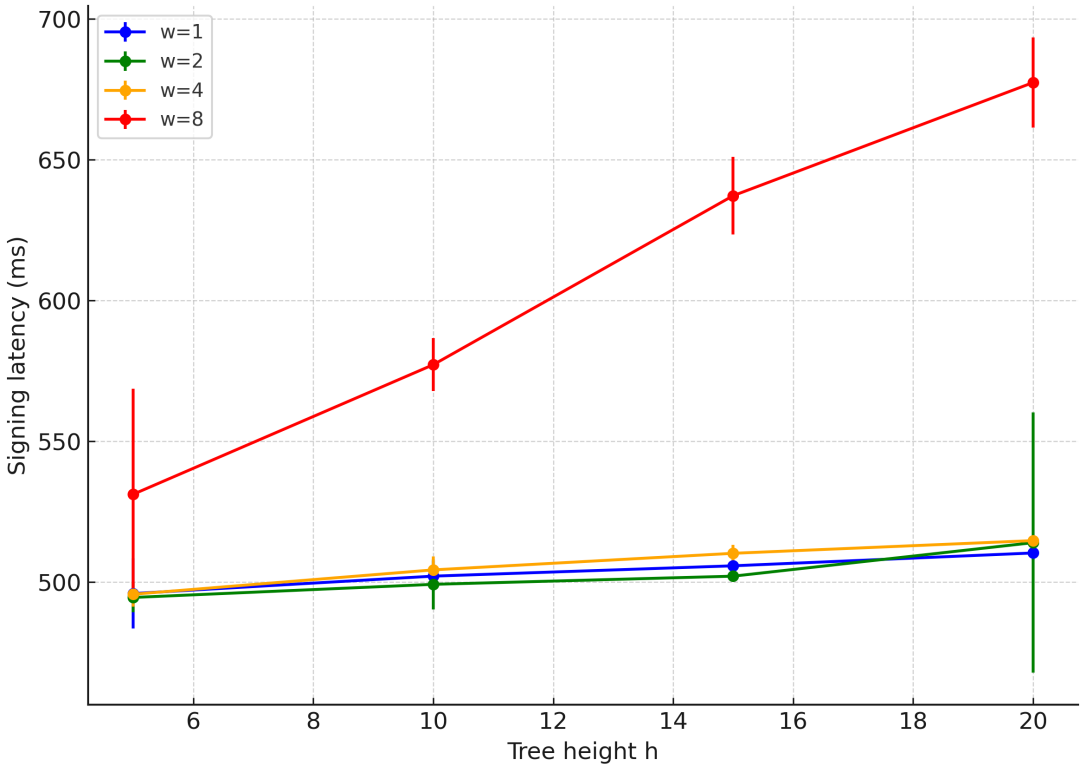


Figure 5.1: Execution time of LMS signing (average of 30 rounds) in the software-only implementation, with error bars indicating ±1 std. dev.

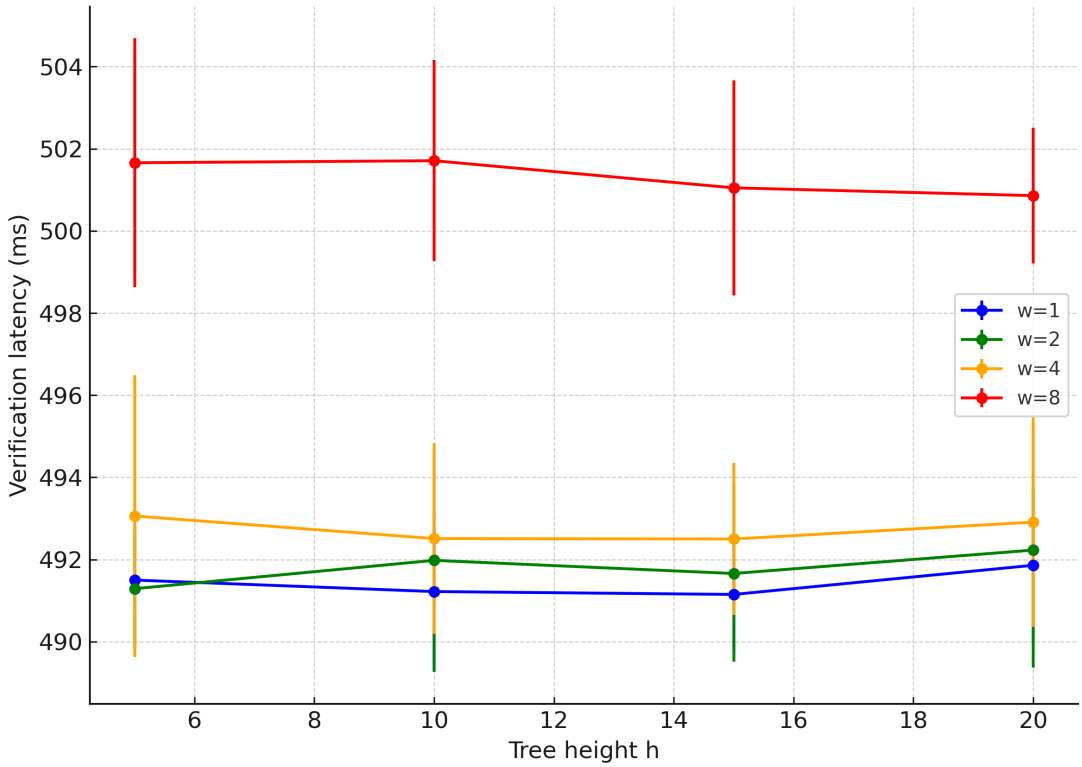


Figure 5.2: Execution time of LMS verification (average of 30 rounds) in the software-only implementation, with error bars indicating ±1 std. dev.

5.3.3 Profiling Results

Profiling reports generated with `gprof` were collected for all parameter sets, with particular focus on the case $h = 20$, which provides a representative trade-off between computational intensity and practicality. To process the profiling output, a custom Python script (Appendix A.6.1) was used to parse the `gprof` tables, filter out functions unrelated to the LMS algorithm (e.g., initialization, zeroization), and aggregate the time spent in SHA-256 related functions versus other auxiliary routines. The script then produced stacked bar plots normalized to 100% of execution time.

Figures 5.3–5.5 report the results for `keygen`, `sign`, and `verify` respectively, across all Winternitz parameters $w \in \{1,2,4,8\}$ at $h = 20$. In both key generation and signing, hashing clearly dominates the runtime, consuming over 80% of the total execution time in all cases. This confirms that SHA-256 is the computational bottleneck of LMS.

By contrast, the stacked bar for verification (Figure 5.5) appears empty. This is not due to a lack of operations, but rather to the extremely short runtime: the

verification phase completes so quickly that `gprof` cannot accumulate measurable percentages for any function. The appendix (see Table B.4 and related) still reports the raw call counts, which confirm that the expected hash invocations take place. The absence of measurable bars therefore illustrates the lightweight nature of verification compared to key generation and signing.

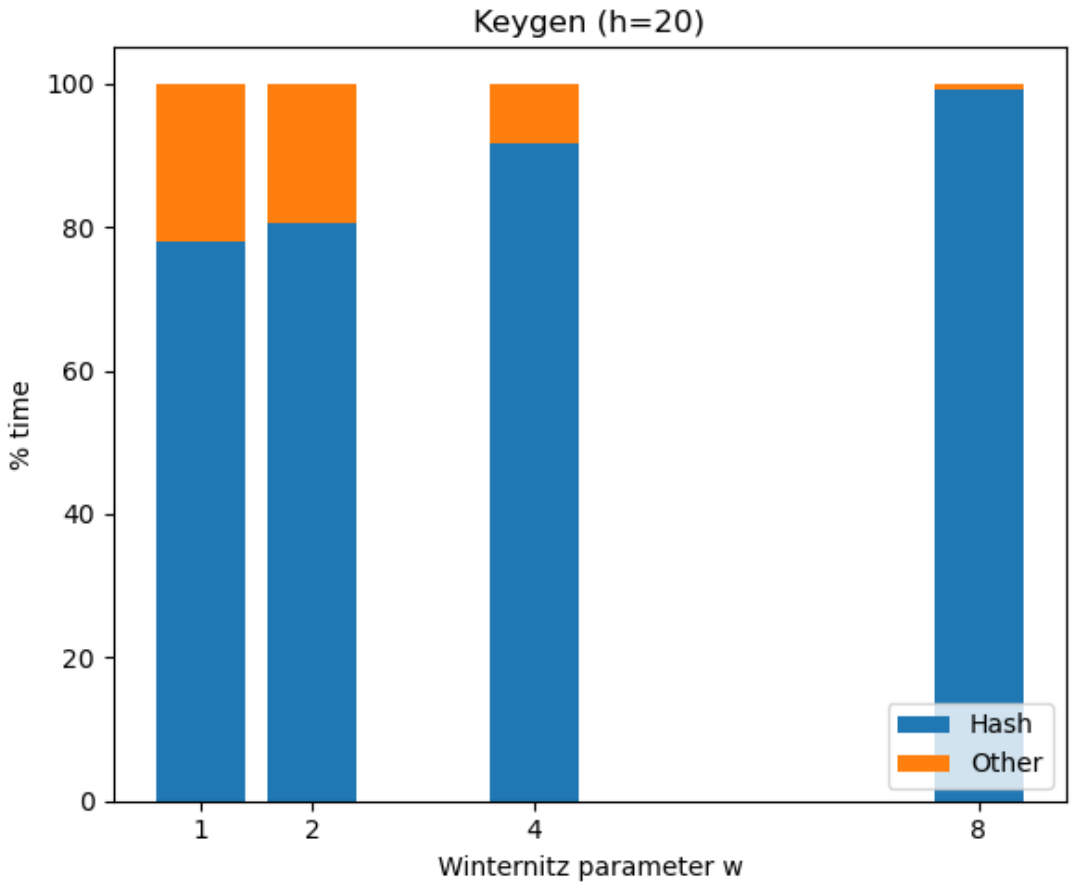


Figure 5.3: Profiling result for LMS key generation at $h = 20$, showing the proportion of runtime spent in hash-related functions versus other operations.

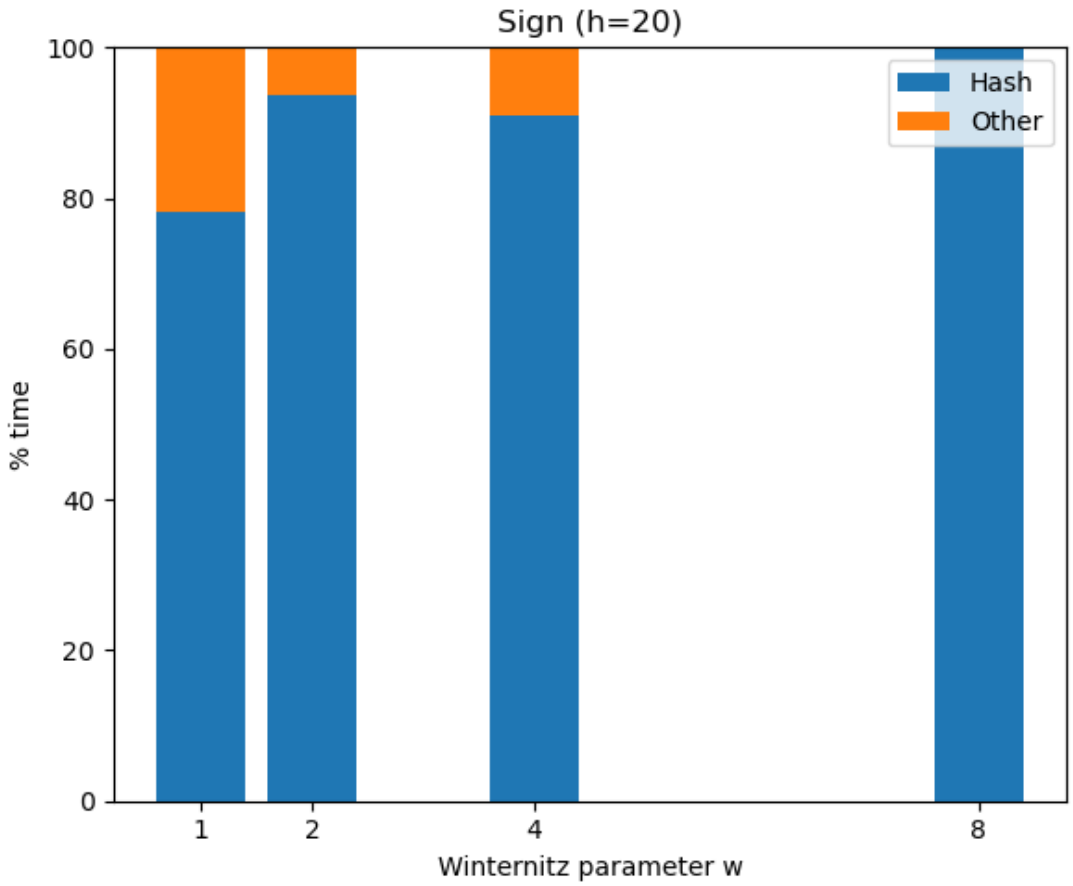


Figure 5.4: Profiling result for LMS signing at $h = 20$, showing the dominance of SHA-256 operations.

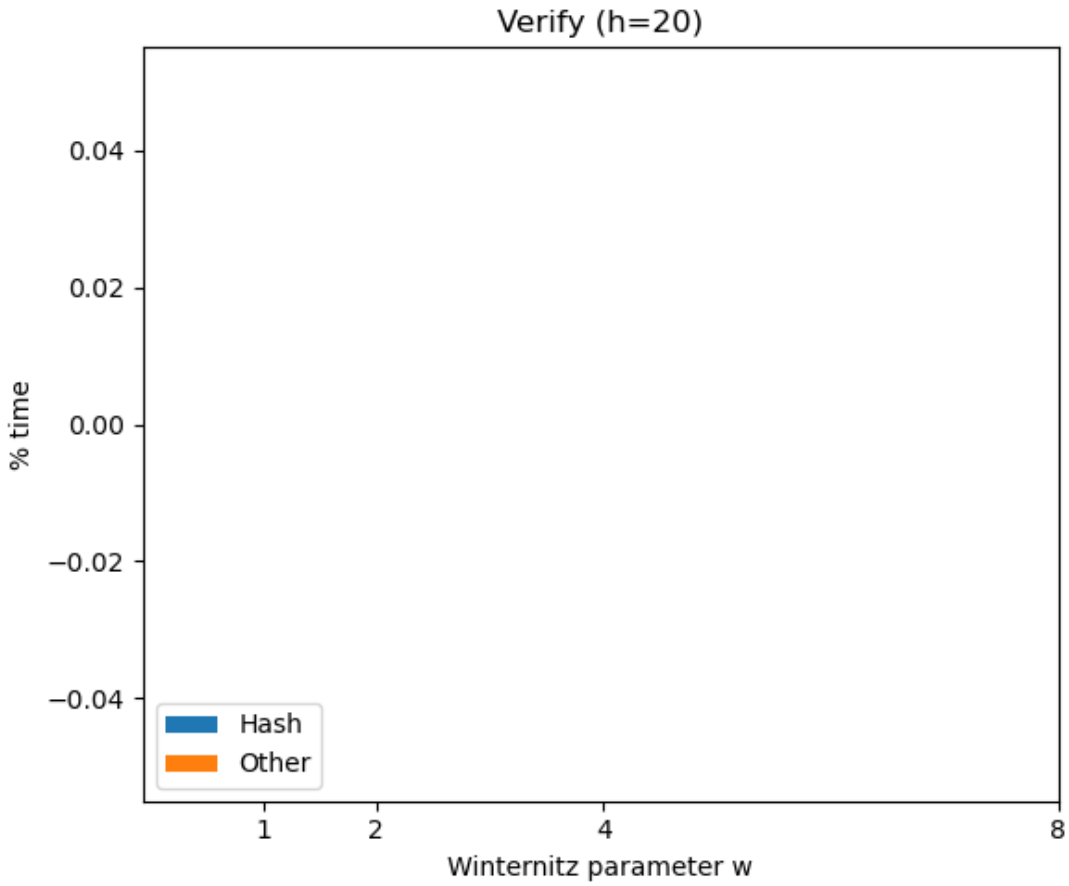


Figure 5.5: Profiling result for LMS verification at $h = 20$. The stacked bar appears empty due to the very short runtime, confirming the negligible cost of verification.

The figures confirm that SHA-256 dominates all phases of the scheme. Depending on the operation and the value of w, hash-related functions account for more than 80–95% of the runtime, leaving only a small fraction to encoding or structural operations. This observation provides a strong motivation for the hardware/software co-design explored in the next section, where SHA-256 invocations are offloaded to the accelerator.

5.4 HW–SW Co-Design Benchmarking

The second set of experiments evaluated the hardware/software co-design, where SHA-256 operations were offloaded to the custom accelerator instantiated on the

PYNQ-Z2 FPGA fabric. The benchmarking methodology initially mirrored the software-only setup: LMS operations were executed for the NIST-recommended parameter sets, and runtime was measured with `clock_gettime()`, while profiling data was collected with `gprof`.
At first sight, the raw execution times were slower than in the pure software implementation. Since key generation is by far the most time-consuming phase, it was omitted from this stage of testing to avoid days of runtime. The focus was placed instead on the `sign` and `verify` operations, which can be executed in a few seconds and are therefore better suited to iterative performance analysis.

5.4.1 Initial Results

The raw numerical results for all parameter sets are reported in Appendix B.2.1. For $h = 25$, key generation was performed on the HPC cluster for completeness, but signing and verification attempts on the PYNQ-Z2 again failed with `hss_error_out_of_memory`. Figures 5.6 and 5.7 report the execution times of signing and verification when SHA-256 calls are offloaded to the accelerator. The measurements demonstrate that, despite successful integration of the accelerator, the overall measured execution time was longer than in the software-only case.

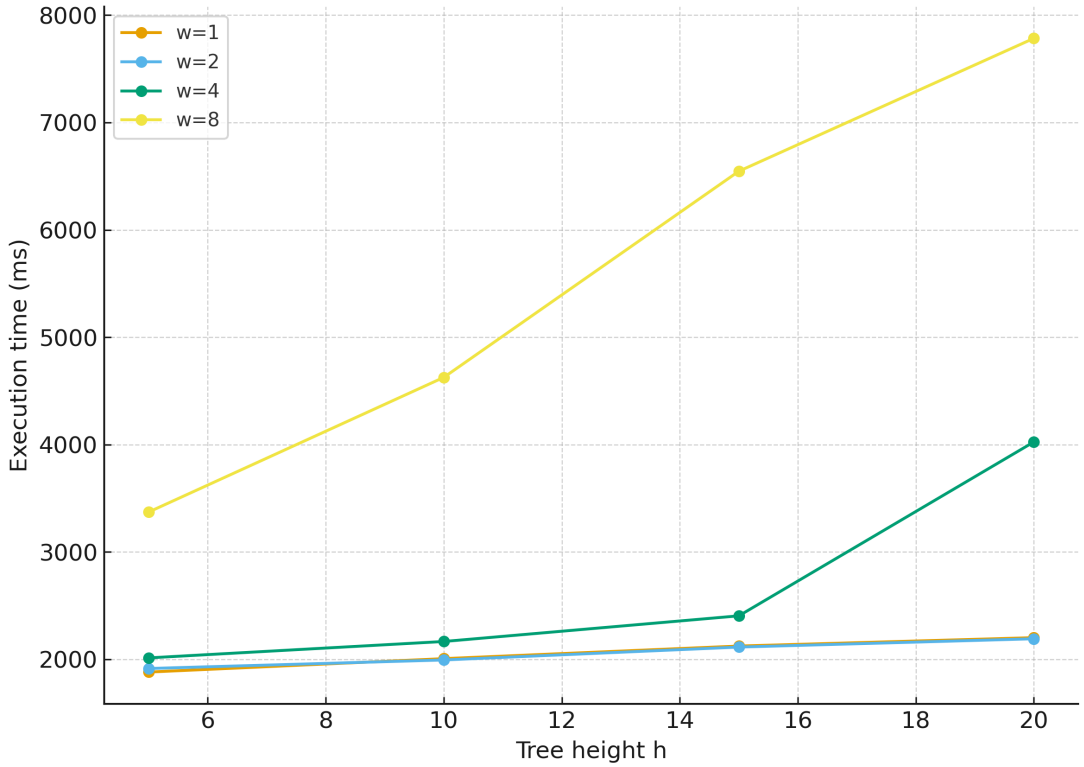


Figure 5.6: Execution time of LMS signing in the HW–SW co-design, before driver optimization.

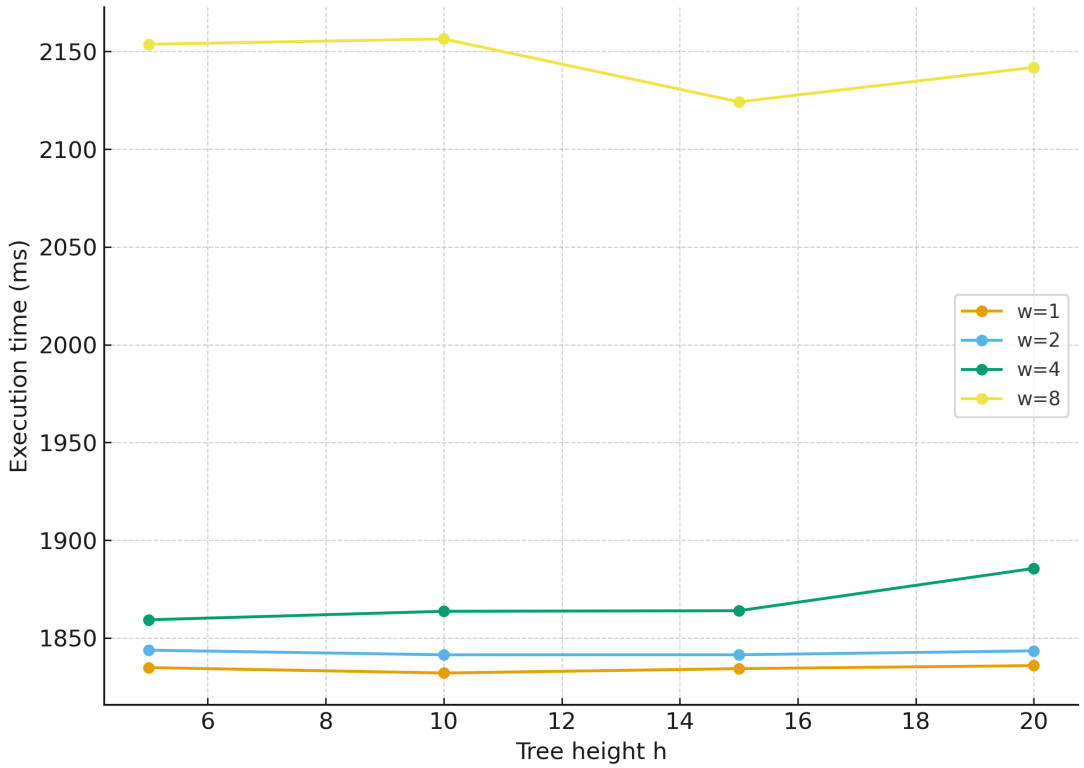


Figure 5.7: Execution time of LMS verification in the HW–SW co-design, before driver optimization.

Profiling of the HW/SW co–design reveals a clear height–dependent split. For small trees (h=5–10), the runtime is dominated by `SHA256_Update`, i.e., the incremental software path that the current accelerator cannot offload. As h grows, the number of single–shot hashes explodes and the time attributed to `hw_hash` (the wrapper that drives the accelerator) increases accordingly; by h=20 it becomes the single largest contributor in most (h, w) configurations. Importantly, the time charged to `hw_hash` is not CPU computation; it includes the cost of marshalling data, padding, register I/O/polling, and waiting for the device to complete. In other words, the bottleneck shifts from pure hashing (software) to offload overhead and device latency/clock frequency (hardware). In practice, this also highlights a scalability issue: the accelerator offers little benefit for small workloads, where software routines dominate, and becomes the main cost driver only at large h, where its overhead outweighs the gains from hardware acceleration. A per–configuration summary is reported in Table 5.4.

h	$w = 1$	$w = 2$	$w = 4$	$w = 8$
5	1.50 / 75.19	1.56 / 73.44	0.78 / 77.52	4.23 / 66.20
10	2.03 / 79.31	1.50 / 75.19	2.28 / 81.68	5.96 / 67.55
15	11.37 / 56.87	9.84 / 60.62	25.00 / 43.53	51.25 / 14.62
20	29.52 / 29.27	37.16 / 23.90	53.39 / 11.67	62.80 / 1.32

Table 5.4: Relative time distribution (%) between `hw_hash` and `SHA256_Update` for signing (HW/SW co–design).

This observation explains why the wall-clock runtime did not improve despite the per-cycle advantage of the hardware core. The bottleneck has not been removed, but relocated from software hashing routines to the communication and synchronization path between software and hardware. The disappointing outcome of these initial benchmarks highlighted that raw acceleration alone is insufficient: the surrounding driver and software stack must also be carefully optimized. The next subsection therefore investigates the impact of driver design on performance.

5.4.2 Driver Optimization Results

To quantify the benefit of the driver redesign, the performance of the unoptimised and optimised implementations was compared. The unoptimised driver was evaluated with a single measurement per configuration, since the runtimes were consistently high and further repetition was unnecessary. In contrast, the optimised driver was benchmarked over 30 rounds, with averages and standard deviations reported to improve statistical robustness.
Table 5.5 reports the comparison, showing the average latency with the optimised driver alongside the one-shot baseline from the unoptimised driver. A speedup factor was then computed as the ratio of the two. Figure 5.8 visualises the same results.

h	w	Unoptimised (1 run)	Optimised (avg ± std)	Speedup
5	1	1880.469	1847.387 ± 2.156	1.02×
10	1	2005.605	1892.182 ± 23.684	1.06×
15	1	2122.998	1927.240 ± 1.711	1.10×
20	1	2201.208	1954.115 ± 2.466	1.13×
5	2	1913.687	1858.488 ± 3.006	1.03×
10	2	1993.180	1883.553 ± 1.997	1.06×
15	2	2113.278	1921.105 ± 2.461	1.10×
20	2	2190.628	1945.469 ± 2.194	1.13×
5	4	2012.551	1885.732 ± 2.057	1.07×
10	4	2165.307	1936.780 ± 2.202	1.12×
15	4	2404.307	2006.931 ± 3.106	1.20×
20	4	4024.542	2055.915 ± 2.189	1.96×
5	8	3371.347	2320.352 ± 3.973	1.45×
10	8	4625.536	2689.925 ± 2.143	1.72×
15	8	6546.042	3272.497 ± 2.194	2.00×
20	8	7784.278	3661.065 ± 2.814	2.13×

Table 5.5: Comparison of unoptimised vs. optimised driver (signature latency in ms). Optimised results are averages over 30 rounds with ±1 std. dev. Speedup is computed as the ratio of the two.

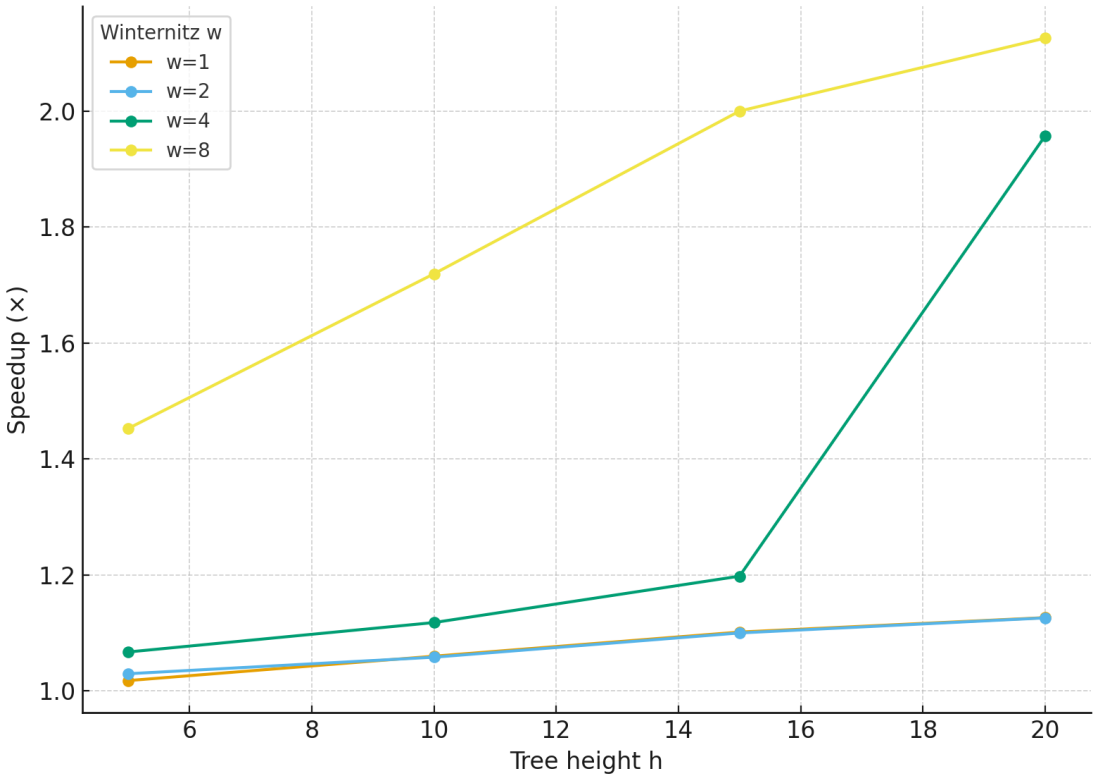


Figure 5.8: Driver optimisation speedup (signature generation)

The results show a clear trend: for small Winternitz parameters (w=1,2), the impact of driver optimisation is limited (1.02–1.13×). In contrast, as w increases, the overhead of accelerator invocation grows and the optimised driver brings substantial benefits. At w=8, the speedup exceeds 2×, demonstrating that software overhead, rather than the accelerator core itself, was the dominant performance bottleneck. Verification latencies showed negligible differences between the two driver versions, confirming that this operation is lightweight and not bottlenecked by driver overhead. For completeness, the raw verification results over 30 rounds are reported in Appendix B.2.2.

Despite the observed gains, the optimised HW/SW co–design still lags behind the pure software implementation. Even at its best configuration (h=20, w=8), signature generation with the accelerator remains significantly slower than the software-only baseline. This counter–intuitive outcome shows that removing driver overhead was not sufficient to unlock a net speedup. To clarify the root cause, a deeper investigation of the underlying execution characteristics is required, which is the focus of the next section.

5.5 Cycle-Level Analysis

The driver optimisation experiments showed that performance improved considerably compared to the initial implementation, yet the HW/SW co–design still lagged behind the pure software baseline in wall–clock time. To clarify the root cause, a cycle–level analysis was performed to quantify the intrinsic cost of a single SHA-256 invocation in hardware compared to software.

5.5.1 Intrinsic Hardware Latency

To measure the cycle count of the accelerator in isolation, the VHDL testbench of the SHA-256 core was modified to include a simple cycle counter. The counter started when the `msg_valid` signal was asserted and stopped once `hash_valid` was raised. The instrumented testbench is reported in Appendix A.6.2.
Simulation in Vivado reported that one SHA-256 operation requires a fixed latency of 137 clock cycles. This figure represents the intrinsic datapath cost of the hardware core, independent of driver or software overhead. It provides a theoretical lower bound for the number of cycles required by the programmable logic to process one message block.

5.5.2 Measurement Methodology

The next step was to measure the effective cycle cost of invoking the hash function from C, both in pure software (OpenSSL) and through the hardware wrapper (`hw_hash`). A dedicated microbenchmark, shown in Appendix A.6.3, was developed, invoking either the SW or HW path for a controlled number of iterations. The number of iterations was parameterized at compile time (`ITERATIONS=1,10,100,1000`) to investigate scaling behavior and to reduce the relative impact of fixed overheads. For each configuration, the benchmark was executed 50 times to obtain stable statistical estimates. The `perf stat` tool was used to collect the number of retired CPU cycles during each execution, and the results were appended to log files for later processing. The entire process was automated through the shell script shown in Appendix A.6.4, which builds both hardware and software versions of the benchmark, runs the desired number of rounds, and collects cycle counts.

5.5.3 Results

Table 5.6 reports the average number of retired cycles measured over 50 repetitions for both HW and SW modes, for iteration counts of 1, 10, 100, and 1000. Standard deviations are also shown to illustrate measurement stability.

Iterations	HW Cycles	SW Cycles
1	2,089,897 ± 67,428	4,283,441 ± 100,828
10	2,349,323 ± 67,730	4,269,507 ± 80,369
100	4,695,088 ± 92,729	4,509,441 ± 94,171
1000	28,100,721 ± 356,813	6,769,845 ± 98,718

Table 5.6: Average CPU cycles (mean ± standard deviation) over 50 runs.

Figure 5.9 provides a per-iteration bar comparison between HW and SW modes, with error bars corresponding to one standard deviation. For 1 and 10 iterations, the HW path consistently requires approximately half the number of cycles of the SW path. This matches expectations: the hardware accelerator offloads the hash computation almost entirely, resulting in fewer retired CPU cycles.

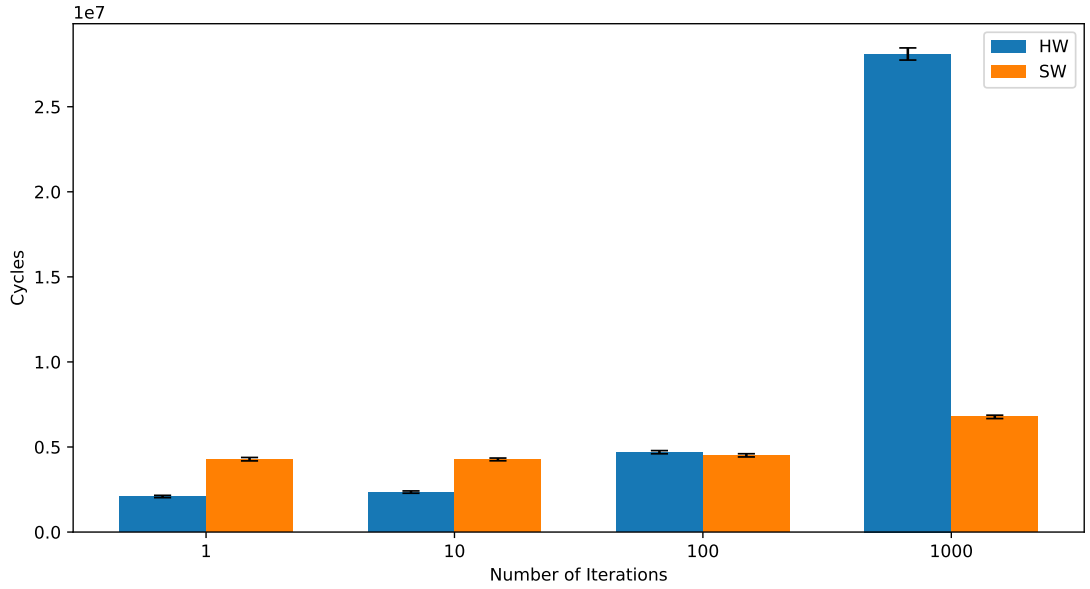


Figure 5.9: Average retired cycles for HW vs. SW modes at different iteration counts (mean ± std).

Figure 5.10 shows the same data on a logarithmic x-axis to highlight scaling trends. While the SW mode grows almost linearly with the number of iterations, the HW mode exhibits a sharp increase beyond 100 iterations. At 1000 iterations, the HW path requires more than four times the number of CPU cycles than SW, reversing

the advantage observed at smaller scales.

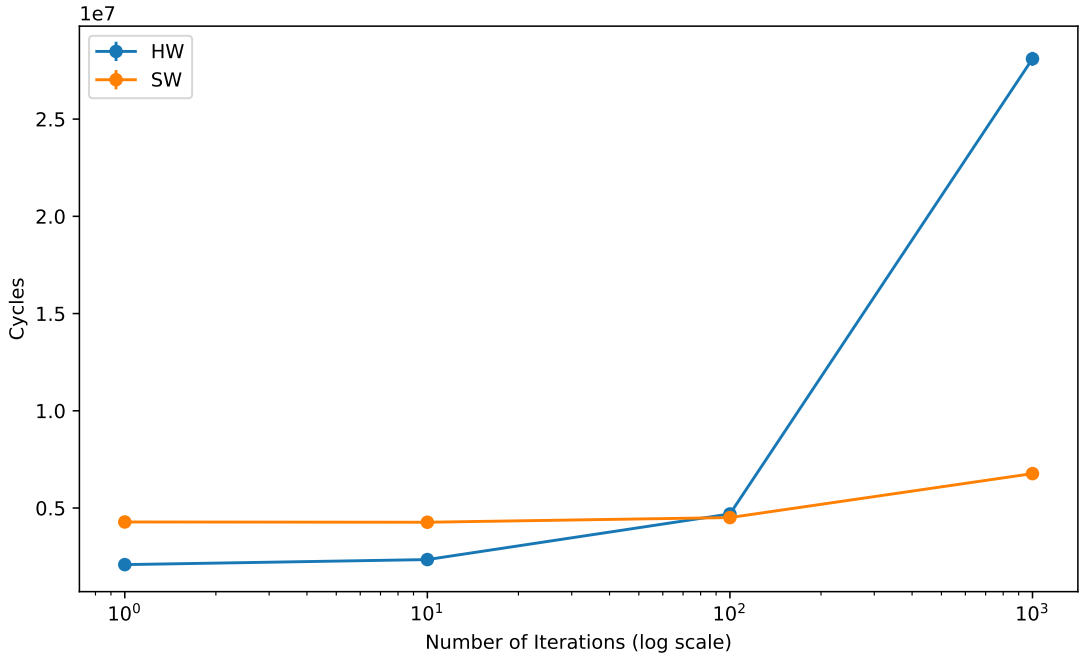


Figure 5.10: Scaling of HW and SW cycle counts with the number of iterations (logarithmic x-axis).

5.5.4 Discussion

These results offer a nuanced picture of the HW/SW trade-off at the cycle level: for short bursts of work, involving one or ten hash invocations, the HW/SW path requires roughly half the number of CPU cycles compared to the pure software path. This behaviour matches expectations for an efficient hardware offload: the CPU issues the command, then waits while the accelerator performs the computation, retiring relatively few instructions in the process.
As the workload grows to intermediate sizes (around 100 iterations), the initial advantage of the hardware path largely disappears. At this scale, the fixed overhead of each accelerator invocation begins to accumulate, effectively cancelling out the cycle savings achieved by offloading the hash computations to hardware.
For large workloads (1000 iterations), the situation reverses completely: the HW/SW path performs substantially worse than the software baseline in terms of total CPU cycles. This is a direct consequence of the current driver model, which invokes the accelerator once per iteration, incurring the cost of AXI transactions,

input padding, and synchronization on every call. Meanwhile, the software loop executes entirely within the OpenSSL library, benefiting from tight, highly optimized instruction loops without any system call overhead.
In essence, the hardware accelerator is *per-call cycle efficient*, but the current driver model introduces a significant fixed cost per invocation. This explains the non-linear behavior observed at larger iteration counts. Two main factors contribute to this:

(i) The accelerator runs at a much lower frequency (31.25 MHz) compared to the ARM cores, so raw cycle reductions do not translate directly to time savings.

(ii) The blocking driver model serializes CPU and PL activity, preventing any overlap and making per-call synchronization overhead dominant at scale.

To unlock the full potential of the accelerator, batching multiple messages into a single offload, using DMA transfers, or switching to an interrupt-driven model would be required. These optimizations could reduce per-call overhead and make the hardware solution competitive at higher workloads.

5.6 Break-Even Accelerator Frequency Estimation

The cycle-level analysis demonstrated that the SHA-256 hardware core is intrinsically more cycle-efficient than software, but this advantage does not currently translate into a wall-clock speedup. A natural question is therefore: at what clock frequency would the accelerator need to run to outperform the software implementation? This section derives an estimate of such a break-even frequency under both optimistic and more realistic assumptions.

5.6.1 Performance Model

The wall-clock time of a single hash invocation through the HW/SW co-design can be modelled as

$$T_{\mathrm{HW/SW}} \;=\; T_{\mathrm{SW(pre)}} + T_{\mathrm{HW}} + T_{\mathrm{SW(post)}} \;\approx\; 2T_{\mathrm{driver}} + T_{\mathrm{HW}}, \tag{5.1}$$

where T_{driver} represents the fixed software cost of preparing the input, performing AXI register I/O, and synchronising with the device, while T_{HW} is the actual accelerator latency.
For N consecutive hash invocations, the total time becomes

$$T_{\mathrm{HW/SW}}(N) \;\approx\; 2T_{\mathrm{driver}} + N \cdot T_{\mathrm{iter}}, \tag{5.2}$$

where T_{iter} is the per-hash iteration cost. Assuming the software baseline has total time $T_{\text{SW}}(N)$, the *break-even frequency* $f^{\star}_{\text{PL}}$ for the accelerator is the value at which

$$T_{\text{HW/SW}}(N; f^{\star}_{\text{PL}}) = T_{\text{SW}}(N). \tag{5.3}$$

5.6.2 Optimistic Model

As a first approximation, we assume that the entire per-iteration cost T_{iter} scales inversely with the programmable logic (PL) clock frequency. Letting $f_{\text{PL,now}}$ denote the current PL frequency and $T_{\text{iter,now}}$ the measured per-iteration time at that frequency, the required break-even frequency is

$$f^{\star}_{\text{PL}} \;\approx\; f_{\text{PL,now}} \cdot \frac{N\, T_{\text{iter,now}}}{T_{\text{SW}}(N) - 2T_{\text{driver}}}. \tag{5.4}$$

5.6.3 Realistic Model

In practice, not all of T_{iter} scales with frequency. A large portion is due to fixed software and I/O overhead, while only the core hashing latency (137 cycles) depends on the PL clock. We therefore refine the model by splitting

$$T_{\text{iter}} = T_{\text{fixed}} + \frac{C_{\text{core}}}{f_{\text{PL}}}, \tag{5.5}$$

where T_{fixed} is the per-call overhead independent of PL frequency, and C_{core} is the number of accelerator cycles per hash (137 for the implemented core). Substituting (5.5) into (5.2) and solving for $f^{\star}_{\text{PL}}$ yields

$$f^{\star}_{\text{PL}} = \frac{C_{\text{core}}\, N}{T_{\text{SW}}(N) - 2T_{\text{driver}} - NT_{\text{fixed}}}. \tag{5.6}$$

This situation corresponds to a regime in which the combined cost of driver invocation and per-iteration software overhead already exceeds the total runtime of the software baseline, even assuming an infinitely fast accelerator. In other words, the overhead alone is sufficient to make the HW/SW path slower than software. Under these conditions, increasing the PL clock frequency cannot compensate for the structural overheads: the only way to make acceleration beneficial is to reduce the fixed costs associated with each invocation.
It is important to distinguish between the roles of T_{driver} and T_{fixed} in this model. The former represents a *one-time* cost per benchmark call: it includes all pre- and post-processing steps performed outside the hashing loop, such as interface initialisation, memory mapping, and other fixed software setup routines. This is why it appears in (5.2) as $2T_{\text{driver}}$ rather than being multiplied by the number of iterations N.

By contrast, T_{fixed} captures the *per-iteration* software and I/O overhead that does not scale with the PL clock frequency. This includes operations such as writing message blocks to AXI registers, applying padding and length encoding, issuing control signals, and polling for completion. These steps are repeated for each hash invocation inside the loop, which is why T_{fixed} is multiplied by N in (5.6). Distinguishing between these two components is essential: grouping them together would either underestimate overheads for large N (if treated as purely driver cost) or overestimate them for small N (if treated as purely per-iteration cost).

5.6.4 Numerical Estimates

To extract the parameters T_{driver} and T_{fixed} from the measured cycle counts, a simple linear model was fitted to the hardware microbenchmark results. Let `HW_cycles`(N) denote the average number of retired CPU cycles measured with `perf` for the hardware path when hashing N blocks. The measurements follow the relation

$$\texttt{HW\_cycles}(N) \;\approx\; A \;+\; B\,N, \tag{5.7}$$

where A corresponds to the one-time driver cost (enter/exit) and B captures the per-iteration cost.
Converting cycles to time using the CPU frequency $f_{\text{CPU}} = 650$ MHz gives

$$T_{\text{driver}} = \frac{A}{2\,f_{\text{CPU}}}, \tag{5.8}$$

$$T_{\text{iter,now}} = \frac{B}{f_{\text{CPU}}}, \tag{5.9}$$

where $T_{\text{iter,now}}$ represents the total per-iteration cost measured at the current PL clock frequency.
This per-iteration cost is then decomposed into a frequency-independent software/I/O component and a frequency-dependent core latency:

$$T_{\text{iter,now}} = T_{\text{fixed}} + \frac{C_{\text{core}}}{f_{\text{PL,now}}}, \tag{5.10}$$

where $C_{\text{core}} = 137$ is the number of accelerator cycles per hash, and $f_{\text{PL,now}} = 31.25$ MHz is the current PL frequency. Solving for T_{fixed} yields

$$T_{\text{fixed}} = T_{\text{iter,now}} - \frac{C_{\text{core}}}{f_{\text{PL,now}}}. \tag{5.11}$$

Using the measured data from Section 5.5, the linear fit produced $A \approx 2.08 \times 10^6$ cycles and $B \approx 2.60 \times 10^4$ cycles per iteration. This gives

$$T_{\text{driver}} \approx 1.60 \text{ ms}, \qquad T_{\text{iter,now}} \approx 40.0\ \mu\text{s}, \qquad T_{\text{fixed}} \approx 35.6\ \mu\text{s}.$$

The latter corresponds to the per-iteration cost not attributable to the 137-cycle core latency (≈ 4.4 μs at 31.25 MHz).
Table 5.7 reports the resulting break-even PL frequencies for $N = 1, 10, 100, 1000$ iterations, using both the optimistic and realistic models.

Table 5.7: Estimated break-even PL frequencies $f_{\mathrm{PL}}^{\star}$ for different iteration counts.

Iterations	Optimistic Model	Realistic Model
1	∼33 MHz	∼36 MHz
10	∼33 MHz	∼40 MHz
100	∼33 MHz	∼60 MHz
1000	∼173 MHz	*no finite solution*

5.6.5 Discussion

The estimates reveal two important insights. First, for short to moderate workloads (up to tens of hashes), a modest increase of the PL clock to around 50–60 MHz would be sufficient to break even with the software baseline. This is plausible on the PYNQ-Z2 platform and would already yield speedups without architectural changes.
Second, for larger workloads (hundreds to thousands of hashes), simply raising the PL frequency is not enough. The per-call overhead dominates the total runtime, making the denominator in (5.6) negative: in other words, no clock frequency can compensate for the current driver design. This explains why the HW/SW co-design performs worse than software at $N = 1000$ despite the intrinsic efficiency of the SHA-256 core.
The model therefore highlights two complementary optimisation paths: increasing the accelerator frequency can yield immediate gains for short workloads, while batching, DMA transfers, or interrupt-driven drivers are required to reduce T_{fixed} and T_{driver} to unlock scalability.

5.7 Pre-Quantum Baseline: ECDSA–SHA256

To contextualize the performance of LMS on embedded hardware, a set of reference measurements was carried out using the classical ECDSA–P256 digital signature scheme combined with SHA-256. This represents a widely deployed pre-quantum baseline: ECDSA–P256 is currently used in numerous secure-boot frameworks and firmware signing pipelines in industry. Comparing LMS against this well-known scheme provides a practical sense of the performance gap that must be bridged when transitioning to post-quantum signature schemes.

5.7.1 Methodology

Firmware signing and verification were implemented using OpenSSL's EVP API for ECDSA over the NIST P-256 curve. The benchmarking script is shown in Appendix A.6.5. A firmware file (`firmware.bin`), identical to the one used in the LMS benchmarks, was signed and verified to ensure consistency. Key pairs were generated once and stored in PEM format; signing and verification benchmarks were then performed using the same pre-generated key in order to isolate the cost of the cryptographic operations from key generation.
Signing and verification were each repeated over 30 rounds. The benchmarking harness used `clock_gettime()` with the `CLOCK_MONOTONIC` clock source to measure wall-clock time, as in the LMS measurements. The average time and standard deviation over the 30 rounds were computed for both signing and verification. Additionally, the signature length was recorded to compare the size overhead between the two schemes.

5.7.2 Results

Table 5.8 reports the average signing and verification times for ECDSA–P256 on the PYNQ-Z2, measured over 30 rounds using the EVP API. Signature size is also included for completeness. This table represents the classical, pre-quantum baseline against which the various LMS configurations are compared.

	Sign (ms)	Verify (ms)
ECDSA P-256 (EVP)	426.15 ± 4.83	419.28 ± 5.04

Table 5.8: ECDSA–P256 baseline on PYNQ-Z2 (30 rounds).

Representative LMS Comparison

To avoid overwhelming the reader with the full parameter space, a selected set of LMS parameter configurations is presented in Table 5.9. These configurations are chosen to illustrate the trade-offs between runtime and signature size across different (h, w) combinations, covering small, medium, and large parameter sets, as well as the hardware/software co-design at the most demanding point.

Scheme	Params	Sign (ms)	Verify (ms)	Sig. size (B)	Sign / ECDSA	Verify / ECDSA
ECDSA P-256 (EVP)	–	426	419	64	1.0	1.0
LMS (SW)	(h=10, w=1)	502	491	∼ 8,832	1.18×	1.17×
LMS (SW)	(h=20, w=4)	515	493	∼ 2,816	1.21×	1.18×
LMS (SW)	(h=20, w=8)	677	501	1,112	1.59×	1.20×
LMS (HW/SW)	(h=20, w=8)	3661	∼ 1945	1,112	8.6×	4.6×

Table 5.9: Comparison between ECDSA–P256 and selected LMS parameter sets on PYNQ-Z2. Ratios are relative to ECDSA (lower is better).

Figures 5.11 and 5.12 compare the signing and verification latencies of all LMS parameter sets against a fixed ECDSA P-256 baseline on the PYNQ-Z2 board. The figures show the evolution of execution time as a function of the Winternitz parameter w, with separate curves for each tree height $h \in \{5,10,15,20\}$. Error bars represent ±1 standard deviation over 30 measurement rounds. The ECDSA baseline is shown as a horizontal dashed line for reference.

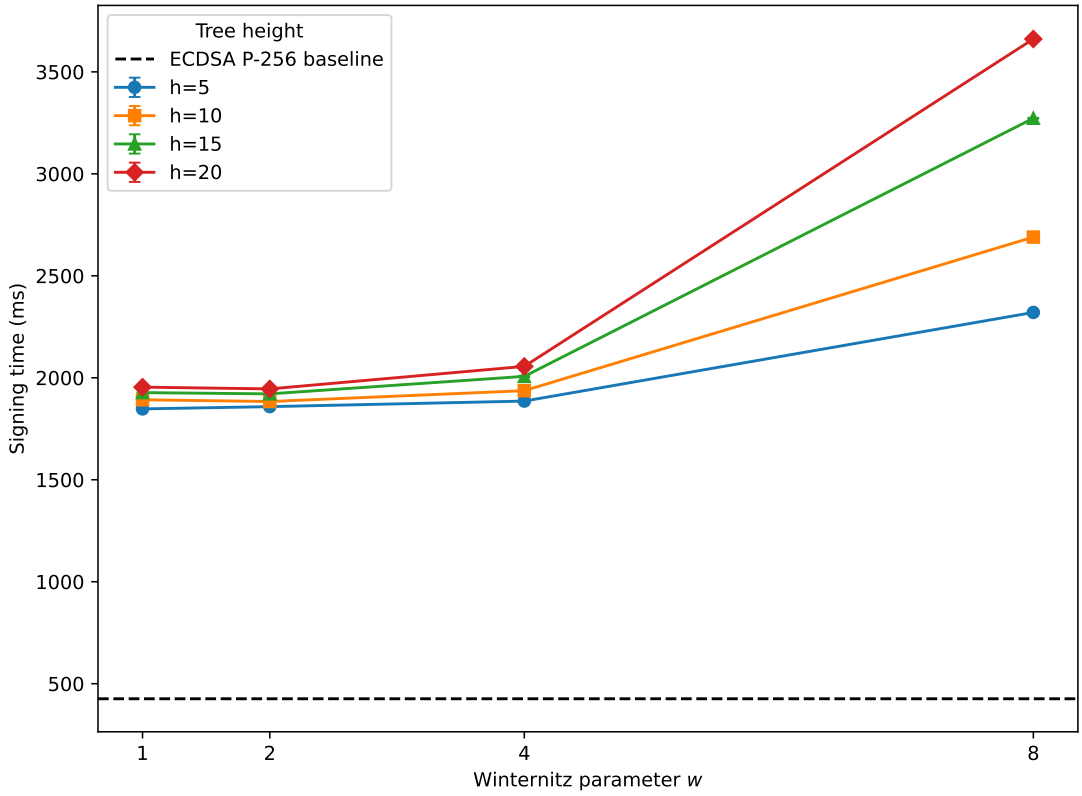


Figure 5.11: LMS signing time across Winternitz parameters and tree heights compared to the ECDSA P-256 baseline (dashed). Error bars show ±1 standard deviation.

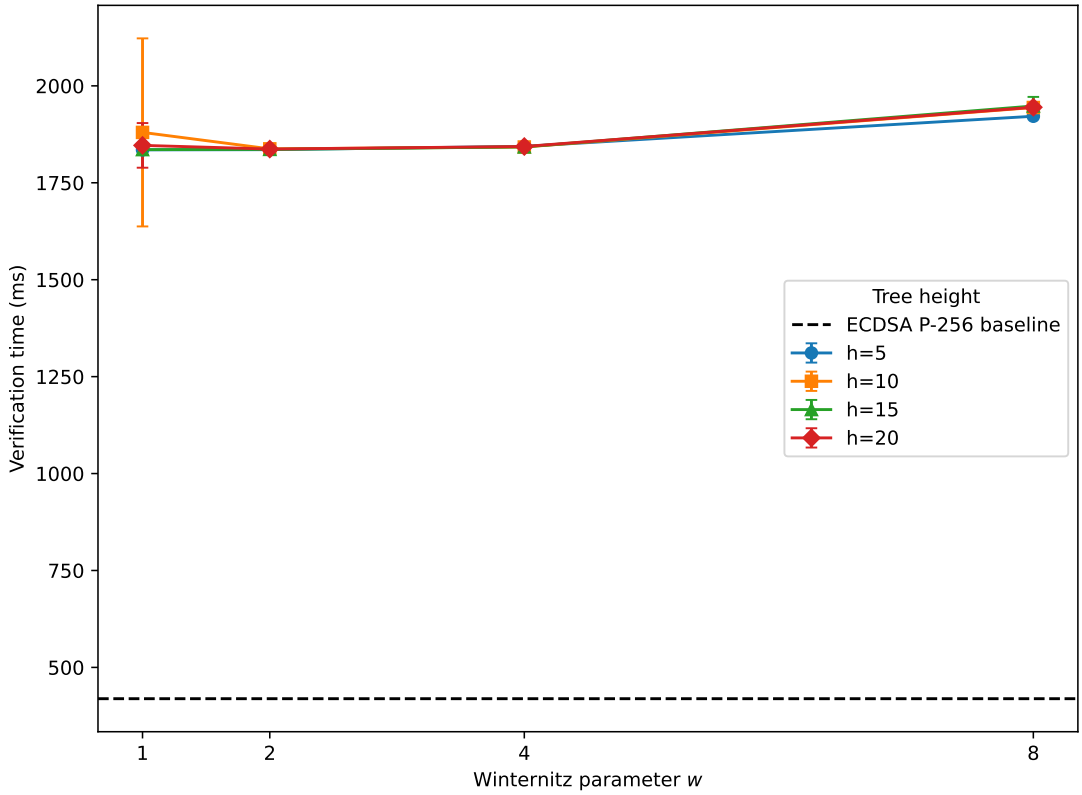


Figure 5.12: LMS verification time across Winternitz parameters and tree heights compared to the ECDSA P-256 baseline (dashed).

Unlike ECDSA, which corresponds to a single, highly optimized operating point, LMS spans a wide design space parameterized by (h, w). The plots clearly show that, on the PYNQ-Z2 platform, even the smallest LMS configurations are several times slower than ECDSA for both signing and verification. Signing cost increases significantly with w and h, with the largest configurations (w=8, h=20) taking several seconds per operation. Verification times are much less sensitive to (h, w), forming nearly flat curves across Winternitz parameters, but still remain above the ECDSA baseline.

5.8 Limitations and Future Work

5.8.1 Limitations

The results presented in this thesis should be interpreted in light of a number of methodological, platform, implementation, and scope limitations that influenced

both the design choices and the measured performance.

Methodological Limitations

The benchmarking campaign focused exclusively on runtime performance, in terms of execution time and retired cycles, and did not include measurements of power consumption or energy efficiency, which are essential metrics for evaluating post-quantum schemes on embedded platforms. All measurements were conducted on a single hardware platform (PYNQ-Z2), which limits the generalizability of the results to other architectures such as STM32 MCUs, automotive SoCs, or ASIC-based designs. Moreover, the analysis considered only SHA-256 as the underlying hash function; alternative NIST-approved hash variants (e.g., SHA-256/192 or SHAKE functions) were not explored.
All benchmarks were performed in a single-threaded configuration, without exploiting the potential for parallelism or pipelining in LMS key generation or signing. In addition, the LMS software implementation used in this work is based on the official RFC reference code, which is functionally correct but not optimized for performance. At the time of this work, OpenSSL had not yet released an LMS implementation for key generation, signing, and verification. Consequently, the software baseline does not reflect the level of optimization that can be expected from future mainstream cryptographic libraries. Repeating this benchmarking campaign once an optimized OpenSSL LMS backend becomes available would likely yield more representative software baselines.

Platform Limitations

The programmable logic on the PYNQ-Z2 board operates at a relatively low frequency of 31.25 MHz, which strongly affects the computed break-even frequencies and the overall performance of the accelerator. The accelerator is accessed through a blocking AXI-Lite interface, which prevents overlapping CPU and PL activity and increases per-call latency. No DMA or interrupt-driven mechanisms were employed; instead, data transfers rely on CPU-driven register writes and polling, introducing significant overhead. Furthermore, the platform hosts only a single accelerator instance, and no multi-accelerator or multithreaded architectures were explored.

Implementation Limitations

The SHA-256 accelerator implements only the compression function for single message blocks and does not support incremental hash operations in hardware. As a result, only a fraction of the total hash computations are offloaded, particularly during LMS signing and verification, where incremental hash updates are dominant. The driver, although optimized to remove syscall overhead via memory mapping,

follows a per-block invocation model in which each hash block triggers a separate sequence of data marshalling, padding, AXI register transactions, and polling.
The LMS implementation itself is single-threaded and follows the RFC reference algorithm without structural optimizations such as caching of internal nodes or batching of hash computations. The accelerator is limited to a single datapath, without pipelining, message chaining, or parallel hashing capabilities. Architectural techniques such as batched hash processing or Merkle tree computations in hardware were not implemented, which limits the achievable performance.

Scope Limitations

The scope of this work is limited to performance evaluation; security analysis, side-channel resistance, and formal verification of the hardware/software stack were not investigated. Within the performance analysis, only signing and verification operations were benchmarked in the HW/SW co-design; key generation was excluded due to its prohibitive runtime at high tree heights. No end-to-end secure boot prototype was developed and tested; the reported measurements are based on isolated microbenchmarks. Finally, the ECDSA comparison was limited to a single P-256 software baseline, without including hardware-accelerated ECDSA implementations or alternative elliptic curve parameters.

5.8.2 Future Work

Several directions can be pursued to address these limitations and extend this work. From a platform perspective, increasing the accelerator clock frequency, enabling DMA or interrupt-driven operation, and instantiating multiple accelerators could significantly reduce driver overhead and improve throughput. At the algorithmic level, supporting incremental hashing in hardware, batching multiple message blocks per invocation, or pipelining hash computations would allow the accelerator to target the dominant `SHA256_Update` phase of LMS signing and verification, which is currently handled in software.
On the software side, the availability of a native OpenSSL LMS implementation would provide a more realistic and optimized software baseline for comparison. Integrating the accelerator into a full secure-boot chain would enable end-to-end performance and security evaluation. Finally, extending the benchmarking to additional platforms, evaluating energy consumption, and exploring ASIC or high-frequency FPGA targets would provide valuable insight into the practical feasibility of LMS for post-quantum secure boot in real embedded systems.

Chapter 6

Conclusions

This work investigated the integration of post-quantum digital signature algorithms within the secure boot process of real-time embedded systems. The motivation stems from the imminent advent of quantum computers, which threatens currently deployed public-key cryptography, and from the need to adopt standardized, quantum-resistant algorithms in constrained environments such as automotive electronic control units.

An initial comparative analysis of standardized post-quantum signature schemes identified hash-based signatures, and in particular the Leighton–Micali Signature (LMS) scheme, as a promising candidate. LMS offers strong security guarantees grounded solely on the hardness of preimage and collision resistance of the underlying hash function, compact and fixed key sizes, and a mature standardization and deployment profile. While lattice-based schemes such as Falcon or ML-DSA achieve more compact signatures, their mathematical complexity and implementation challenges make LMS a more practical choice for early-stage adoption in embedded secure boot frameworks.

The core of this thesis consisted of a detailed software–hardware co-design and benchmarking study of LMS on a Xilinx PYNQ-Z2 board. The LMS algorithm was implemented in software following the IETF reference code, modified to run in a single-core, single-threaded setting to enable accurate and reproducible measurements. A SHA-256 hardware accelerator was ported to the platform and integrated with the processing system through an AXI-Lite interface. A custom Linux character driver was then developed and optimized to minimize overhead through memory-mapped I/O, reducing system call latency and maximizing the effective throughput of the accelerator.

Extensive benchmarking campaigns were performed on both software-only and hardware-accelerated configurations, across multiple LMS parameter sets and over 30 independent measurement rounds. The experiments measured execution times, cycle counts, and profiling information for key generation, signature generation, and

verification. The results highlighted the dominant role of hashing in LMS signing and verification, justifying the acceleration strategy. Driver optimization led to a substantial reduction of per-call overhead, while intrinsic accelerator latency was measured through a cycle-level testbench. Break-even analysis was then conducted to estimate the minimum accelerator frequency required to outperform pure software hashing. Finally, the hardware-accelerated LMS performance was compared against a pre-quantum ECDSA–SHA256 baseline to provide a practical deployment perspective.

The study demonstrated that, although the current single-accelerator configuration with AXI-Lite access cannot yet surpass optimized software hashing for short messages, the methodology enables precise identification of performance bottlenecks and provides a clear path toward scalable acceleration. The results also confirm that LMS, despite higher signature generation costs compared to ECDSA, remains a viable candidate for post-quantum secure boot thanks to its simplicity, predictability, and hardware-friendly structure.

Looking forward, several avenues of improvement have been identified. These include adopting multi-accelerator or pipelined designs to enable parallel hashing, moving from AXI-Lite to higher-throughput interfaces, exploiting DMA for bulk message transfer, and integrating LMS implementations from optimized cryptographic libraries such as OpenSSL once available. Beyond performance, integrating LMS into a complete secure boot chain on heterogeneous SoCs would provide further insights into system-level impacts and required optimizations.

In conclusion, this thesis contributes a complete and reproducible evaluation framework for post-quantum LMS signatures on embedded platforms, combining algorithmic analysis, hardware–software co-design, and quantitative benchmarking. The results offer both a baseline and a methodology for future research toward efficient, quantum-safe secure boot in resource-constrained environments.

Appendix A

Source Code and Benchmark Data

A.1 Optimized SHA-256 Driver

This section reports the complete source code of the optimized SHA-256 Linux character device driver used to interface with the accelerator. The driver employs `mmap()`-based register access to minimize system call overhead and achieve low-latency hash operations.

```
// sha256_driver.c     minimal char driver for SHA-256
   accelerator on PYNQ

#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/io.h>
#include <linux/uaccess.h>
#include <linux/mm.h>

#define DRIVER_NAME "sha256"
#define DEVICE_NAME "sha256"
#define CLASS_NAME  "sha256_class"

#define BASE_ADDR  0x40000000
#define MEM_SIZE   0x1000      // 4 KB mapped size

MODULE_LICENSE("GPL");
```

```
20 MODULE_AUTHOR("Francesco Corvaglia");
21 MODULE_DESCRIPTION("Minimal SHA-256 driver for PYNQ-Z2 Board
   ");
22 MODULE_VERSION("0.2");
23
24 static dev_t dev_number;
25 static struct cdev sha256_cdev;
26 static struct class *sha256_class;
27
28 static void __iomem *base_addr;
29
30 static int sha256_open(struct inode *inode, struct file *
   file) {
31     return 0;
32 }
33
34 static int sha256_release(struct inode *inode, struct file *
   file) {
35     return 0;
36 }
37
38 static ssize_t sha256_read(struct file *file, char __user *
   buf, size_t len, loff_t *offset) {
39     uint32_t value;
40
41     if (*offset + sizeof(uint32_t) > MEM_SIZE)
42         return -EINVAL;
43
44     value = ioread32(base_addr + *offset);
45
46     if (copy_to_user(buf, &value, sizeof(uint32_t)))
47         return -EFAULT;
48
49     return sizeof(uint32_t);
50 }
51
52 static ssize_t sha256_write(struct file *file, const char
   __user *buf, size_t len, loff_t *offset) {
53     uint32_t value;
54
55     if (len != sizeof(uint32_t) || *offset + sizeof(uint32_t
   ) > MEM_SIZE)
56         return -EINVAL;
57
58     if (copy_from_user(&value, buf, sizeof(uint32_t)))
```

```
59          return -EFAULT;
60
61      iowrite32(value, base_addr + *offset);
62
63      return sizeof(uint32_t);
64  }
65
66  static int sha256_mmap(struct file *file, struct
       vm_area_struct *vma) {
67      unsigned long phys = BASE_ADDR;
68      unsigned long vsize = vma->vm_end - vma->vm_start;
69
70      if (vsize > MEM_SIZE)
71          return -EINVAL;
72
73      // Mark memory as non-cached
74      vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
75
76      // Map the physical address to user space
77      return (remap_pfn_range(vma,
78                              vma->vm_start,
79                              phys >> PAGE_SHIFT,
80                              vsize,
81                              vma->vm_page_prot));
82  }
83
84
85  static const struct file_operations fops = {
86      .owner = THIS_MODULE,
87      .open = sha256_open,
88      .release = sha256_release,
89      .read = sha256_read,
90      .write = sha256_write,
91      .mmap = sha256_mmap,
92  };
93
94  static int __init sha256_init(void) {
95      int ret;
96
97      // Reserve the region
98      if (!request_mem_region(BASE_ADDR, MEM_SIZE, DRIVER_NAME
       )) {
99          pr_err("Unable to reserve memory region\n");
100         return -EBUSY;
101     }
```

```
102
103     base_addr = ioremap(BASE_ADDR, MEM_SIZE);
104     if (!base_addr) {
105         pr_err("Unable to map memory\n");
106         release_mem_region(BASE_ADDR, MEM_SIZE);
107         return -ENOMEM;
108     }
109
110     // Allocate char device number
111     ret = alloc_chrdev_region(&dev_number, 0, 1, DEVICE_NAME
    );
112     if (ret < 0) {
113         pr_err("Failed to allocate char device region\n");
114         goto err_unmap;
115     }
116
117     cdev_init(&sha256_cdev, &fops);
118     sha256_cdev.owner = THIS_MODULE;
119
120     ret = cdev_add(&sha256_cdev, dev_number, 1);
121     if (ret < 0) {
122         pr_err("Failed to add char device\n");
123         goto err_unregister;
124     }
125
126     sha256_class = class_create(THIS_MODULE, CLASS_NAME);
127     if (IS_ERR(sha256_class)) {
128         pr_err("Failed to create device class\n");
129         ret = PTR_ERR(sha256_class);
130         goto err_cdev_del;
131     }
132
133     if (device_create(sha256_class, NULL, dev_number, NULL,
    DEVICE_NAME) == NULL) {
134         pr_err("Failed to create device\n");
135         ret = -EINVAL;
136         goto err_class;
137     }
138
139     pr_info("SHA256 driver loaded. /dev/%s ready.\n",
    DEVICE_NAME);
140     return 0;
141
142 err_class:
143     class_destroy(sha256_class);
```

```
144 err_cdev_del:
145     cdev_del(&sha256_cdev);
146 err_unregister:
147     unregister_chrdev_region(dev_number, 1);
148 err_unmap:
149     iounmap(base_addr);
150     release_mem_region(BASE_ADDR, MEM_SIZE);
151     return ret;
152 }
153
154 static void __exit sha256_exit(void) {
155     device_destroy(sha256_class, dev_number);
156     class_destroy(sha256_class);
157     cdev_del(&sha256_cdev);
158     unregister_chrdev_region(dev_number, 1);
159     iounmap(base_addr);
160     release_mem_region(BASE_ADDR, MEM_SIZE);
161     pr_info("SHA256 driver unloaded.\n");
162 }
163
164 module_init(sha256_init);
165 module_exit(sha256_exit);
```

Listing A.1: Optimized SHA-256 driver (full source).

A.2 Python Overlay Testbench

To validate the correct functionality of the SHA-256 accelerator via PYNQ overlays, a dedicated Python testbench was developed. It loads the hardware overlay, applies SHA-256 message padding, transfers input words to the accelerator, and compares the resulting digest with reference values computed using Python's `hashlib`.

```
1  from pynq import Overlay
2  import struct
3  import time
4
5  # Load the overlay and IP block
6  overlay = Overlay("/home/xilinx/Tesi/sha256_tests/
       sha256_wrapper.bit")
7  sha256_accel = overlay.sha256_ctrl_axi_0
8
9  # Padding functions
10 def create_binary(input_string):
```

```
11      return ''.join(format(ord(x), '08b') for x in
    input_string)
12
13  def append_bit_1(input_string):
14      return input_string + '1'
15
16  def append_k_bit(input_string):
17      while len(input_string) % 512 != 448:
18          input_string += '0'
19      return input_string
20
21  def append_length_of_message(input_string):
22      length_bin = format(len(input_string), '064b')
23      return length_bin
24
25  def pad_message(input_string):
26      binary = create_binary(input_string)
27      padded = append_k_bit(append_bit_1(binary)) +
    append_length_of_message(binary)
28      return padded
29
30  # Function to write one 32-bit word
31  def write_word(word_bits, is_last):
32      word = int(word_bits, 2)
33      while sha256_accel.read(0x4) & 0x1 == 0:  # wait for
    msg_ready
34          pass
35      if is_last:
36          sha256_accel.write(0x4, 0x8)  # set msg_last
37      sha256_accel.write(0x0, word)  # write to data_reg
38
39  # Feed message and get hash
40  def hash_message(padded_binary):
41      num_words = len(padded_binary) // 32
42      for i in range(num_words):
43          word = padded_binary[i*32:(i+1)*32]
44          is_last = (i == num_words - 1)
45          write_word(word, is_last)
46
47      while sha256_accel.read(0x4) & 0x2 == 0:  # wait for
    hash_valid
48          pass
49
50      hash_words = []
51      for offset in range(0x8, 0x28, 4):  # h0 to h7
```

```
52          print(f"Reading offset {offset}")
53          h = sha256_accel.read(offset)
54          hash_words.append(h)
55
56      sha256_accel.write(0x4, 0x4)  # set hash_ack
57      return hash_words
58
59  # Main loop
60  if __name__ == "__main__":
61      while True:
62          input_str = input("Enter message to hash (type 'Bye'
    to exit): ")
63          if input_str == "Bye":
64              print("Exiting...")
65              break
66
67          padded = pad_message(input_str)
68          print("Padded Hexadecimal Message:")
69          padded_hex = ''.join(format(int(padded[i:i+8], 2), '
   02x') for i in range(0, len(padded), 8))
70          print(padded_hex)
71          print(len(padded_hex))
72          result = hash_message(padded)
73
74          print("SHA-256 Hash:")
75          print(''.join(format(word, '08x') for word in result
   ))
76          print()
```

Listing A.2: Python testbench for functional validation of the SHA-256 accelerator.

A.3 C Testbench for SHA-256 Accelerator

A standalone C testbench was used to validate the accelerator without relying on the PYNQ framework. The testbench consists of a header file, the accelerator access implementation, and a `main` program that executes test vectors and checks the computed digests.

A.3.1 Header File

This header declares constants, register addresses, and function prototypes for interacting with the accelerator.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <errno.h>



void write_reg(int fd, off_t offset, uint32_t value);

void sha256_acc_init();

void sha256_acc_close();

uint32_t read_reg(int fd, off_t offset);

void wait_for_ready(int fd);

void wait_for_hash_valid(int fd);

void print_hash(int fd);

static void pad_message(const uint8_t *message, size_t
    msg_len, uint8_t **padded_msg, size_t *padded_len);

void hw_hash(const void *message, size_t message_len, void *
    result);
```

Listing A.3: Header file for the C SHA-256 accelerator testbench.

A.3.2 Testbench Implementation

The implementation file provides functions to transfer data, start computations, and read back results from the accelerator.

```c
#include "sha256_acc_tb.h"
#include <pthread.h>

#define DEVICE_FILE "/dev/sha256"

// Register offsets
#define DATA_REG    0x00
```

```
8  #define STATUS_REG   0x04
9  #define H0_REG       0x08
10
11 // Status bits
12 #define MSG_READY    (1 << 0)
13 #define HASH_VALID   (1 << 1)
14 #define HASH_ACK     (1 << 2)
15 #define MSG_LAST     (1 << 3)
16
17 // SHA-256 constants
18 #define BLOCK_BITS 512
19 #define BLOCK_BYTES 64
20 #define WORD_BYTES 4
21 #define WORDS_IN_BLOCK 16
22 #define LEN_BITS 64
23
24
25 static int sha256_fd = -1;
26
27
28 void sha256_acc_init() {
29     if (sha256_fd < 0) {
30         sha256_fd = open(DEVICE_FILE, O_RDWR);
31         if (sha256_fd < 0) {
32             perror("Failed to open SHA256 device");
33             exit(1);
34         }
35     }
36 }
37
38 void sha256_acc_close() {
39     if (sha256_fd >= 0) {
40         close(sha256_fd);
41         sha256_fd = -1;
42     }
43 }
44
45
46
47 void write_reg(int fd, off_t offset, uint32_t value) {
48     if (pwrite(fd, &value, sizeof(value), offset) != sizeof(
    value)) {
49         fprintf(stderr, "Write failed at %lx: %s\n",
50                 (unsigned long)offset, strerror(errno));
51         exit(1);
```

```
52      }
53  }
54
55  uint32_t read_reg(int fd, off_t offset) {
56      uint32_t value;
57      if (pread(fd, &value, sizeof(value), offset) != sizeof(
        value)) {
58          fprintf(stderr, "Read failed at %lx: %s\n",
59                  (unsigned long)offset, strerror(errno));
60          exit(1);
61      }
62      return value;
63  }
64
65  void wait_for_ready(int fd) {
66      while ((read_reg(fd, STATUS_REG) & MSG_READY) == 0) {
67    }
68  }
69
70  void wait_for_hash_valid(int fd) {
71      while ((read_reg(fd, STATUS_REG) & HASH_VALID) == 0) {
72    }
73  }
74
75  void print_hash(int fd) {
76      printf("Hash registers:\n");
77      for (int i = 0; i < 8; i++) {
78          printf("h%d: 0x%08x\n", i, read_reg(fd, H0_REG + i
        *4));
79      //printf("Reading offset %08x\n", H0_REG + i * 4);
80      }
81  }
82
83  static void pad_message(const uint8_t *message, size_t
        msg_len, uint8_t **padded_msg, size_t *padded_len) {
84      size_t num_bits = msg_len * 8;
85      size_t num_blocks = (num_bits + LEN_BITS + 1 ) /
        BLOCK_BITS + 1;
86      *padded_len = num_blocks * BLOCK_BYTES;
87
88      *padded_msg = calloc(1, *padded_len);
89      if (!*padded_msg) {
90          perror("malloc failed");
91          exit(1);
92      }
```

```
93
94    // Copy original message (no need to swap yet as we're
      copying bytes)
95      memcpy(*padded_msg, message, msg_len);
96
97      // Append '1' bit (0x80) - this is a byte operation, no
      swapping needed
98      (*padded_msg)[msg_len] = 0x80;
99
100     // Append length in bits as 64-bit big-endian value
101     uint64_t length = num_bits;
102     // Convert to big-endian before copying
103     uint64_t be_length = __builtin_bswap64(length);
104     memcpy(*padded_msg + *padded_len - 8, &be_length, 8);
105
106     // Now convert the entire message to big-endian words
107     uint32_t *words = (uint32_t *)*padded_msg;
108     for (size_t i = 0; i < (*padded_len / 4); i++) {
109         words[i] = __builtin_bswap32(words[i]);
110     }
111
112     // Print the padded message
113     printf("Padded message (%zu bytes):\n", *padded_len);
114     for (size_t i = 0; i < *padded_len; i++) {
115         printf("%02x ", (*padded_msg)[i]);
116         if ((i + 1) % 16 == 0) printf("\n");
117     }
118     printf("\n");
119 }
120
121 void hw_hash(const void *message, size_t message_len, void *
      result) {
122
123     // Reset the device
124     write_reg(sha256_fd, STATUS_REG, HASH_ACK);
125
126   uint8_t *padded_msg = NULL;
127   size_t padded_len;
128   pad_message(message, message_len, &padded_msg, &padded_len
      );
129
130   //printf("Padded message length: %d", padded_len);
131   size_t num_words = padded_len / 4;
132
133     // Write all words
```

```
134     const uint32_t *words = (const uint32_t *)padded_msg;
135   for (int i = 0; i < num_words; i++) {
136         wait_for_ready(sha256_fd);
137 
138     if (i == num_words - 1) {
139             // Set last flag before writing the last word
140             write_reg(sha256_fd, STATUS_REG, MSG_LAST);
141         }
142         write_reg(sha256_fd, DATA_REG, words[i]);
143     }
144 
145     // Wait for hash to be valid
146     wait_for_hash_valid(sha256_fd);
147 
148   print_hash(sha256_fd);
149 
150   // Read hash result into provided buffer
151     uint32_t *hash_result = (uint32_t *)result;
152     for (int i = 0; i < 8; i++) {
153         hash_result[i] = read_reg(sha256_fd, H0_REG + i*4);
154     }
155 
156     // Acknowledge hash
157     write_reg(sha256_fd, STATUS_REG, HASH_ACK);
158 
159   //close(fd);
160 
161   free(padded_msg);
162 
163 }
```

Listing A.4: C testbench implementation (`sha256_acc_tb.c`).

A.3.3 Main Testbench Application

The main program executes the testbench functions with predefined test vectors, reporting the results to the console.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <stdint.h>
5 #include "sha256_acc_tb.h"
6 
7 #define MAX_INPUT_LEN 1024
```

```
8
9  void print_hash_result(const uint32_t *hash) {
10     for (int i = 0; i < 8; i++) {
11         printf("%08x", hash[i]);
12     }
13     printf("\n");
14 }
15
16
17 int main() {
18     sha256_acc_init();
19
20     char input[MAX_INPUT_LEN];
21
22     while (1) {
23         printf("Enter a message (or 0 to quit): ");
24         if (!fgets(input, sizeof(input), stdin)) {
25             break; // EOF or error
26         }
27
28         // Remove newline if present
29         size_t len = strlen(input);
30         if (len > 0 && input[len - 1] == '\n') {
31             input[len - 1] = '\0';
32             len--;
33         }
34
35         if (strcmp(input, "0") == 0) {
36             break;
37         }
38
39         uint32_t hash_result[8] = {0};
40
41         hw_hash(input, len, hash_result);
42
43         printf("SHA-256 hash: ");
44         print_hash_result(hash_result);
45     }
46
47     sha256_acc_close();
48     return 0;
49 }
```

Listing A.5: Main application for the C SHA-256 accelerator testbench (`sha256_testbench.c`).

A.4 SHA-256 Accelerator Interface

The LMS software was modified to use a lightweight interface for hardware-accelerated hashing. The interface, implemented in `sha256_acc.c/.h`, exposes a simple API that performs complete SHA-256 computations through memory-mapped registers.

A.4.1 Header File

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <errno.h>


void sha256_acc_init();

void sha256_acc_close();

void print_hash(int fd);

void pad_message(const uint8_t *message, size_t msg_len,
    uint8_t **padded_msg, size_t *padded_len);

void hw_hash(const void *message, size_t message_len, void *
    result);
```

Listing A.6: SHA-256 accelerator interface header (`sha256_acc.h`).

A.4.2 Source File

```c
#include "sha256_acc.h"
#include <sys/mman.h>

#define DEVICE_FILE "/dev/sha256"

// Register offsets
#define DATA_REG    0x00
#define STATUS_REG  0x04
#define H0_REG      0x08
```

```
10
11 // Status bits
12 #define MSG_READY    (1 << 0)
13 #define HASH_VALID   (1 << 1)
14 #define HASH_ACK     (1 << 2)
15 #define MSG_LAST     (1 << 3)
16
17 // SHA-256 constants
18 #define BLOCK_BITS 512
19 #define BLOCK_BYTES 64
20 #define WORD_BYTES 4
21 #define WORDS_IN_BLOCK 16
22 #define LEN_BITS 64
23
24 #define MAP_SIZE  0x1000
25
26
27 #define REG(offset) (*(volatile uint32_t *)((uint8_t *)
      sha256_regs + (offset)))
28
29 volatile uint32_t *sha256_regs = NULL;
30
31 static int sha256_fd = -1;
32
33
34 void sha256_acc_init() {
35     if (sha256_fd < 0) {
36         sha256_fd = open(DEVICE_FILE, O_RDWR | O_SYNC);
37         if (sha256_fd < 0) {
38             perror("Failed to open SHA256 device");
39             exit(1);
40         }
41     }
42
43   sha256_regs = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
       MAP_SHARED, sha256_fd, 0);
44     if (sha256_regs == MAP_FAILED) {
45         perror("mmap");
46         exit(1);
47     }
48 }
49
50 void sha256_acc_close() {
51
52   if (sha256_regs) {
```

```
53        munmap((void *)sha256_regs, MAP_SIZE);
54        sha256_regs = NULL;
55    }
56
57    if (sha256_fd >= 0) {
58        close(sha256_fd);
59        sha256_fd = -1;
60    }
61 }
62
63 void print_hash(int fd) {
64    printf("Hash registers:\n");
65    for (int i = 0; i < 8; i++) {
66        printf("h%d: 0x%08x\n", i, REG(H0_REG + i*4));
67    }
68 }
69
70 void pad_message(const uint8_t *message, size_t msg_len,
   uint8_t **padded_msg, size_t *padded_len) {
71    size_t num_bits = msg_len * 8;
72    size_t num_blocks = (num_bits + LEN_BITS + 1) /
   BLOCK_BITS + 1;
73    *padded_len = num_blocks * BLOCK_BYTES;
74
75    *padded_msg = calloc(1, *padded_len);
76    if (!*padded_msg) {
77        perror("malloc failed");
78        exit(1);
79    }
80
81  // Copy original message (no need to swap yet as we're
   copying bytes)
82    memcpy(*padded_msg, message, msg_len);
83
84    // Append '1' bit (0x80) - this is a byte operation, no
   swapping needed
85    (*padded_msg)[msg_len] = 0x80;
86
87    // Append length in bits as 64-bit big-endian value
88    uint64_t length = num_bits;
89    // Convert to big-endian before copying
90    uint64_t be_length = __builtin_bswap64(length);
91    memcpy(*padded_msg + *padded_len - 8, &be_length, 8);
92
93    // Now convert the entire message to big-endian words
```

```
94      uint32_t *words = (uint32_t *)*padded_msg;
95      for (size_t i = 0; i < (*padded_len / 4); i++) {
96          words[i] = __builtin_bswap32(words[i]);
97      }
98
99  }
100
101 void hw_hash(const void *message, size_t message_len, void *
        result) {
102
103     // Reset the device
104   REG(STATUS_REG) = HASH_ACK;
105
106   uint8_t *padded_msg = NULL;
107   size_t padded_len;
108   pad_message(message, message_len, &padded_msg, &padded_len
        );
109   size_t num_words = padded_len / 4;
110
111     // Write all words
112     const uint32_t *words = (const uint32_t *)padded_msg;
113   for (int i = 0; i < num_words; i++) {
114     while ((REG(STATUS_REG) & MSG_READY) == 0);  // wait for
        ready
115       if (i == num_words - 1)
116             REG(STATUS_REG) = MSG_LAST;
117       REG(DATA_REG) = words[i];
118     }
119
120     // Wait for hash to be valid
121     while ((REG(STATUS_REG) & HASH_VALID) == 0);
122
123
124   // Read hash result into provided buffer
125     for (int i = 0; i < 8; i++) {
126
127     uint32_t word = REG(H0_REG + i * 4);
128
129     // Convert from HW endianness to big-endian bytes
130       ((uint8_t *)result)[i * 4 + 0] = (word >> 24) & 0xff;
131       ((uint8_t *)result)[i * 4 + 1] = (word >> 16) & 0xff;
132       ((uint8_t *)result)[i * 4 + 2] = (word >> 8) & 0xff;
133       ((uint8_t *)result)[i * 4 + 3] = (word >> 0) & 0xff;
134
135   }
```

```
136
137     // Acknowledge hash
138   REG(STATUS_REG) = HASH_ACK;
139
140   free(padded_msg);
141
142 }
```

Listing A.7: SHA-256 accelerator interface implementation (`sha256_acc.c`).

A.5 Benchmarking Automation

This section reports the scripts used to automate large-scale LMS benchmarking campaigns, covering all parameter sets and Winternitz values.

A.5.1 Top-Level Benchmark Script

The `benchmark.sh` script recursively executes all LMS benchmarks, creates output directories, and consolidates logs from multiple rounds.

```
1  #!/bin/bash
2
3  # Root path
4  BASE_DIR="./lms_accelerated_benchmark/outputs"
5
6  # Loop over W_X directories
7  for wdir in "$BASE_DIR"/W_*; do
8      echo "Entering $wdir"
9
10     # Convert W_1      W1 (remove underscore)
11   wname=$(basename "$wdir" | tr -d '_')
12
13     # Loop over LMS_SHA256_M32_H{i}_WX directories
14     for bench_dir in $wdir/LMS_SHA256_M32_H*_"$wname"; do
15         echo "Processing $bench_dir"
16
17         # Change into the benchmark directory
18         cd "$bench_dir" || continue
19
20         # Create required directories if not present
21         for dir in  profile_sign profile_verify; do
22             [ -d "$dir" ] || mkdir "$dir"
23         done
24
```

```
        # Clear or create log file
        : > log.txt

        # Run make commands and append output to log.txt
        {
            echo "===== profile_genkey ====="
            make profile_genkey
            echo

            echo "===== profile_sign ====="
            sudo make profile_sign
            echo

            echo "===== profile_verify ====="
            sudo make profile_verify
            echo
        } &>> log.txt

        # Go back to the previous directory silently
        cd - > /dev/null
    done
done
```

Listing A.8: Top-level automation script for LMS benchmarking.

A.5.2 Representative Makefile

Each parameter set directory contains a Makefile that defines rules for profiling LMS key generation, signing, and verification. A representative example is shown below.

```
# Detect absolute path to the directory containing this
   Makefile
MAKEFILE_DIR := $(dir $(abspath $(lastword $(MAKEFILE_LIST))
   ))
PROJECT_ROOT := $(abspath $(MAKEFILE_DIR)/../../..)

AR = /usr/bin/ar
CC = /usr/bin/gcc
CFLAGS = -Wall -O3 -g -pg
OUT = demo
FIRMWARE_PATH = $(PROJECT_ROOT)/../firmware.bin
KEYNAME = Key_ParmSet_10_2
ADVANCE = 1
PARAM_SET = 10/2
```

```
13
14 SRC = $(PROJECT_ROOT)
15
16 all: hss_lib.a \
17      hss_lib_thread.a \
18      hss_verify.a \
19      demo \
20      test_hss
21
22 $(SRC)/hss_lib.a: $(SRC)/hss.o $(SRC)/hss_alloc.o h$(SRC)/
      ss_aux.o $(SRC)/hss_common.o \
23        $(SRC)/hss_compute.o $(SRC)/hss_generate.o $(SRC)/
      hss_keygen.o $(SRC)/hss_param.o $(SRC)/hss_reserve.o \
24        $(SRC)/hss_sign.o $(SRC)/hss_sign_inc.o $(SRC)/
      hss_thread_single.o \
25        $(SRC)/hss_verify.o $(SRC)/hss_verify_inc.o $(SRC)/
      hss_derive.o \
26        $(SRC)/hss_derive.o $(SRC)/hss_zeroize.o $(SRC)/
      lm_common.o \
27        $(SRC)/lm_ots_common.o $(SRC)/lm_ots_sign.o $(SRC)/
      lm_ots_verify.o $(SRC)/lm_verify.o $(SRC)/endian.o \
28        $(SRC)/hash.o $(SRC)/sha256.o
29    $(AR) rcs $@ $^
30
31 $(SRC)/hss_lib_thread.a: $(SRC)/hss.o $(SRC)/hss_alloc.o $(
      SRC)/hss_aux.o $(SRC)/hss_common.o \
32        $(SRC)/hss_compute.o $(SRC)/hss_generate.o $(SRC)/
      hss_keygen.o $(SRC)/hss_param.o $(SRC)/hss_reserve.o \
33        $(SRC)/hss_sign.o $(SRC)/hss_sign_inc.o $(SRC)/
      hss_thread_pthread.o \
34        $(SRC)/hss_verify.o $(SRC)/hss_verify_inc.o \
35        $(SRC)/hss_derive.o $(SRC)/hss_zeroize.o $(SRC)/
      lm_common.o \
36        $(SRC)/lm_ots_common.o $(SRC)/lm_ots_sign.o $(SRC)/
      lm_ots_verify.o $(SRC)/lm_verify.o $(SRC)/endian.o \
37        $(SRC)/hash.o $(SRC)/sha256.o
38    $(AR) rcs $@ $^
39
40 $(SRC)/hss_verify.a: $(SRC)/hss_verify.o $(SRC)/
      hss_verify_inc.o $(SRC)/hss_common.o $(SRC)/
      hss_thread_single.o \
41       $(SRC)/hss_zeroize.o $(SRC)/lm_common.o $(SRC)/
      lm_ots_common.o $(SRC)/lm_ots_verify.o $(SRC)/lm_verify.o
       \
42       $(SRC)/endian.o $(SRC)/hash.o $(SRC)/sha256.o
```

```
43    $(AR) rcs $@ $^
44
45  demo: $(SRC)/demo.c $(SRC)/hss_lib_thread.a
46    $(CC) $(CFLAGS) $(SRC)/demo.c $(SRC)/hss_lib_thread.a -
       lcrypto -lpthread -o $(SRC)/demo
47
48  test_1: $(SRC)/test_1.c lm_ots_common.o lm_ots_sign.o
       lm_ots_verify.o  endian.o hash.o sha256.o hss_zeroize.o
49    $(CC) $(CFLAGS) -o test_1 $(SRC)/test_1.c lm_ots_common.o
       lm_ots_sign.o lm_ots_verify.o  endian.o hash.o sha256.o
       hss_zeroize.o -lcrypto
50
51  test_hss: $(SRC)/test_hss.c $(SRC)/test_hss.h $(SRC)/
       test_testvector.c $(SRC)/test_stat.c \
52            $(SRC)/test_keygen.c $(SRC)/test_load.c $(SRC)/
       test_sign.c $(SRC)/test_sign_inc.c \
53            $(SRC)/test_verify.c $(SRC)/test_verify_inc.c $(
       SRC)/test_keyload.c \
54            $(SRC)/test_reserve.c $(SRC)/test_thread.c $(SRC)/
       test_h25.c $(SRC)/hss.h hss_lib_thread.a
55    $(CC) $(CFLAGS) $(SRC)/test_hss.c $(SRC)/test_testvector.c
        $(SRC)/test_stat.c \
56            $(SRC)/test_keygen.c $(SRC)/test_sign.c $(SRC)/
       test_sign_inc.c $(SRC)/test_load.c \
57            $(SRC)/test_verify.c $(SRC)/test_verify_inc.c $(
       SRC)/test_keyload.c \
58            $(SRC)/test_reserve.c $(SRC)/test_thread.c $(SRC)/
       test_h25.c hss_lib_thread.a \
59            -lcrypto -lpthread -o $(SRC)/test_hss
60
61  hss.o: $(SRC)/hss.c $(SRC)/hss.h $(SRC)/common_defs.h $(SRC)
       /hash.h $(SRC)/endian.h $(SRC)/hss_internal.h $(SRC)/
       hss_aux.h $(SRC)/hss_derive.h
62    $(CC) $(CFLAGS) -c $(SRC)/hss.c -o $(SRC)/$@
63
64  hss_alloc.o: $(SRC)/hss_alloc.c $(SRC)/hss.h $(SRC)/
       hss_internal.h $(SRC)/lm_common.h
65    $(CC) $(CFLAGS) -c $(SRC)/hss_aux.c -o $(SRC)/$@
66
67  hss_aux.o: $(SRC)/hss_aux.c $(SRC)/hss_aux.h $(SRC)/
       hss_internal.h $(SRC)/common_defs.h $(SRC)/lm_common.h $(
       SRC)/endian.h $(SRC)/hash.h
68    $(CC) $(CFLAGS) -c $(SRC)/hss_aux.c -o $(SRC)/$@
69
```

```
70 hss_common.o: $(SRC)/hss_common.c $(SRC)/common_defs.h $(SRC
       )/hss_common.h $(SRC)/lm_common.h
71   $(CC) $(CFLAGS) -c $(SRC)/hss_common.c -o $(SRC)/$@
72
73 hss_compute.o: $(SRC)/hss_compute.c $(SRC)/hss_internal.h $(
       SRC)/hash.h $(SRC)/hss_thread.h $(SRC)/lm_ots_common.h $(
       SRC)/lm_ots.h $(SRC)/endian.h $(SRC)/hss_derive.h
74   $(CC) $(CFLAGS) -c $(SRC)/hss_compute.c -o $(SRC)/$@
75
76 hss_derive.o: $(SRC)/hss_derive.c $(SRC)/hss_derive.h $(SRC)
       /hss_internal.h $(SRC)/hash.h $(SRC)/endian.h
77   $(CC) $(CFLAGS) -c $(SRC)/hss_derive.c -o $(SRC)/$@
78
79 hss_generate.o: $(SRC)/hss_generate.c $(SRC)/hss.h $(SRC)/
       hss_internal.h $(SRC)/hss_aux.h $(SRC)/hash.h $(SRC)/
       hss_thread.h $(SRC)/hss_reserve.h $(SRC)/lm_ots_common.h
       $(SRC)/endian.h
80   $(CC) $(CFLAGS) -c $(SRC)/hss_generate.c -o $(SRC)/$@
81
82 hss_keygen.o: $(SRC)/hss_keygen.c $(SRC)/hss.h $(SRC)/
       common_defs.h $(SRC)/hss_internal.h $(SRC)/hss_aux.h $(
       SRC)/endian.h $(SRC)/hash.h $(SRC)/hss_thread.h $(SRC)/
       lm_common.h $(SRC)/lm_ots_common.h
83   $(CC) $(CFLAGS) -c $(SRC)/hss_keygen.c -o $(SRC)/$@
84
85 hss_param.o: $(SRC)/hss_param.c $(SRC)/hss.h $(SRC)/
       hss_internal.h $(SRC)/endian.h $(SRC)/hss_zeroize.h
86   $(CC) $(CFLAGS) -c $(SRC)/hss_param.c -o $(SRC)/$@
87
88 hss_reserve.o: $(SRC)/hss_reserve.c $(SRC)/common_defs.h $(
       SRC)/hss_internal.h $(SRC)/hss_reserve.h $(SRC)/endian.h
89   $(CC) $(CFLAGS) -c $(SRC)/hss_reserve.c -o $(SRC)/$@
90
91 hss_sign.o: $(SRC)/hss_sign.c $(SRC)/common_defs.h $(SRC)/
       hss.h $(SRC)/hash.h $(SRC)/endian.h $(SRC)/hss_internal.h
        $(SRC)/hss_aux.h $(SRC)/hss_thread.h $(SRC)/hss_reserve.
       h $(SRC)/lm_ots.h $(SRC)/lm_ots_common.h $(SRC)/
       hss_derive.h
92   $(CC) $(CFLAGS) -c $(SRC)/hss_sign.c -o $(SRC)/$@
93
94 hss_sign_inc.o: $(SRC)/hss_sign_inc.c $(SRC)/hss.h $(SRC)/
       common_defs.h $(SRC)/hss.h $(SRC)/hash.h $(SRC)/endian.h
       $(SRC)/hss_internal.h $(SRC)/hss_aux.h $(SRC)/hss_reserve
       .h $(SRC)/hss_derive.h $(SRC)/lm_ots.h $(SRC)/
       lm_ots_common.h $(SRC)/hss_sign_inc.h
```

```
95    $(CC) $(CFLAGS) -c $(SRC)/hss_sign_inc.c -o $(SRC)/$@
96
97  hss_thread_single.o: $(SRC)/hss_thread_single.c $(SRC)/
       hss_thread.h
98    $(CC) $(CFLAGS) -c $(SRC)/hss_thread_single.c -o $(SRC)/$@
99
100 hss_thread_pthread.o: $(SRC)/hss_thread_pthread.c $(SRC)/
       hss_thread.h
101   $(CC) $(CFLAGS) -c $(SRC)/hss_thread_pthread.c -o $(SRC)/
       $@
102
103 hss_verify.o: $(SRC)/hss_verify.c $(SRC)/hss_verify.h $(SRC)
       /common_defs.h $(SRC)/lm_verify.h $(SRC)/lm_common.h $(
       SRC)/lm_ots_verify.h $(SRC)/hash.h $(SRC)/endian.h $(SRC)
       /hss_thread.h
104   $(CC) $(CFLAGS) -c $(SRC)/hss_verify.c -o $(SRC)/$@
105
106 hss_verify_inc.o: $(SRC)/hss_verify_inc.c $(SRC)/
       hss_verify_inc.h $(SRC)/common_defs.h $(SRC)/lm_verify.h
       $(SRC)/lm_common.h $(SRC)/lm_ots_verify.h $(SRC)/hash.h $
       (SRC)/endian.h $(SRC)/hss_thread.h
107   $(CC) $(CFLAGS) -c $(SRC)/hss_verify_inc.c -o $(SRC)/$@
108
109 hss_zeroize.o: $(SRC)/hss_zeroize.c $(SRC)/hss_zeroize.h
110   $(CC) $(CFLAGS) -c $(SRC)/hss_zeroize.c -o $(SRC)/$@
111
112 lm_common.o: $(SRC)/lm_common.c $(SRC)/lm_common.h $(SRC)/
       hash.h $(SRC)/common_defs.h $(SRC)/lm_ots_common.h
113   $(CC) $(CFLAGS) -c $(SRC)/lm_common.c -o $(SRC)/$@
114
115 lm_ots_common.o: $(SRC)/lm_ots_common.c $(SRC)/common_defs.h
        $(SRC)/hash.h
116   $(CC) $(CFLAGS) -c $(SRC)/lm_ots_common.c -o $(SRC)/$@
117
118 lm_ots_sign.o: $(SRC)/lm_ots_sign.c $(SRC)/common_defs.h $(
       SRC)/lm_ots.h $(SRC)/lm_ots_common.h $(SRC)/hash.h $(SRC)
       /endian.h $(SRC)/hss_zeroize.h $(SRC)/hss_derive.h
119   $(CC) $(CFLAGS) -c $(SRC)/lm_ots_sign.c -o $(SRC)/$@
120
121 lm_ots_verify.o: $(SRC)/lm_ots_verify.c $(SRC)/lm_ots_verify
       .h $(SRC)/lm_ots_common.h $(SRC)/hash.h $(SRC)/endian.h $
       (SRC)/common_defs.h
122   $(CC) $(CFLAGS) -c $(SRC)/lm_ots_verify.c -o $(SRC)/$@
123
```

```
124 lm_verify.o: $(SRC)/lm_verify.c $(SRC)/lm_verify.h $(SRC)/
       lm_common.h $(SRC)/lm_ots_common.h $(SRC)/lm_ots_verify.h
        $(SRC)/hash.h $(SRC)/endian.h $(SRC)/common_defs.h
125   $(CC) $(CFLAGS) -c $(SRC)/lm_verify.c -o $(SRC)/$@
126
127 endian.o: $(SRC)/endian.c $(SRC)/endian.h
128   $(CC) $(CFLAGS) -c $(SRC)/endian.c -o $(SRC)/$@
129
130 hash.o: $(SRC)/hash.c $(SRC)/hash.h $(SRC)/sha256.h $(SRC)/
       hss_zeroize.h
131   $(CC) $(CFLAGS) -c $(SRC)/hash.c -o $(SRC)/$@
132
133 sha256.o: $(SRC)/sha256.c $(SRC)/sha256.h $(SRC)/endian.h
134   $(CC) $(CFLAGS) -c $(SRC)/sha256.c -o $(SRC)/$@
135
136
137 # Compilation rules for object files
138 %.o: $(SRC)/%.c
139   $(CC) $(CFLAGS) -c $< -o $@
140
141 clean:
142   -rm $(SRC)/*.o $(SRC)/*.a $(SRC)/demo $(SRC)/test_hss
143
144 run_genkey: $(SRC)/$(OUT)
145   $(PROJECT_ROOT)/$(OUT) genkey $(KEYNAME) $(PARAM_SET)
146
147 profile_genkey: run_genkey
148   gprof $(SRC)/demo gmon.out > ./profile_genkey/
       profile_genkey.txt
149   @echo "Profiling complete. Results saved to profile_genkey
       .txt."
150   mv gmon.out ./profile_genkey/
151
152 run_sign: $(OUT)
153   $(PROJECT_ROOT)/$(OUT) sign $(KEYNAME) $(FIRMWARE_PATH)
154
155 profile_sign: run_sign
156   gprof $(SRC)/demo gmon.out >./profile_sign/profile_sign.
       txt
157   @echo "Profiling complete. Results saved to profile_sign.
       txt."
158   mv gmon.out ./profile_sign/
159
160 run_verify: $(OUT)
161   $(PROJECT_ROOT)/$(OUT) verify $(KEYNAME) $(FIRMWARE_PATH)
```

```
162
163 profile_verify: run_verify
164   gprof $(SRC)/demo gmon.out > ./profile_verify/
       profile_verify.txt
165   @echo "Profiling complete. Results saved to profile_verify
       .txt."
166   mv gmon.out ./profile_verify/
167
168 run_advance: $(OUT)
169   $(PROJECT_ROOT)/$(OUT) advance $(KEYNAME) $(ADVANCE)
170
171 profile_advance: run_advance
172   gprof demo gmon.out > profile_advance.txt
173   @echo "Profiling complete. Results saved to
       profile_advance.txt."
```

Listing A.9: Representative Makefile for profiling LMS operations.

A.6 Benchmark Post-Processing Script

After running 30 independent measurement rounds, a Python script was used to parse the output logs, aggregate signing and verification times per parameter set (w, h), compute averages and standard deviations, and generate a CSV file for plotting and table generation in Chapter 5.

```
1  import os
2  import re
3  import statistics
4  from collections import defaultdict
5
6  # Root folder containing Round1 .. Round30
7  ROOT_DIR = "."
8  OUTPUT_FILE = "benchmark_results.csv"
9
10 # Regex to extract values from log.txt
11 sig_re = re.compile(r"Signature creation time without
       loading key time:\s*([\d.]+)")
12 ver_re = re.compile(r"Verify time:\s*([\d.]+)")
13
14 # Store results
15 results = {}  # results[round][w][h] = {"sig": value, "ver":
       value}
16
17 # For per-(W,H) aggregation
```

```
per_wh_sig = defaultdict(list)
per_wh_ver = defaultdict(list)

for round_num in range(1, 31):
    round_dir = os.path.join(ROOT_DIR, f"Round{round_num}")
    if not os.path.isdir(round_dir):
        continue

    results[round_num] = {}

    for w in [1, 2, 4, 8]:
        w_dir = os.path.join(round_dir, f"W_{w}")
        if not os.path.isdir(w_dir):
            continue

        results[round_num][w] = {}

        for h in [5, 10, 15, 20]:
            h_dir = os.path.join(w_dir, f"LMS_SHA256_M32_H{h}_W{w}")
            log_file = os.path.join(h_dir, "log.txt")

            if not os.path.isfile(log_file):
                continue

            with open(log_file, "r") as f:
                content = f.read()

            sig_match = sig_re.search(content)
            ver_match = ver_re.search(content)

            if sig_match and ver_match:
                sig = float(sig_match.group(1))
                ver = float(ver_match.group(1))
                results[round_num][w][h] = {"sig": sig, "ver": ver}

                # Collect per-(W,H)
                per_wh_sig[(w,h)].append(sig)
                per_wh_ver[(w,h)].append(ver)

# Flatten all results for global stats
all_sig = []
all_ver = []

```

```
with open(OUTPUT_FILE, "w") as out:
    # Raw values
    out.write("Round,W,H,Signature(ms),Verification(ms)\n")
    for r in sorted(results.keys()):
        for w in sorted(results[r].keys()):
            for h in sorted(results[r][w].keys()):
                sig = results[r][w][h]["sig"]
                ver = results[r][w][h]["ver"]
                all_sig.append(sig)
                all_ver.append(ver)
                out.write(f"{r},{w},{h},{sig},{ver}\n")

    # Global summary
    out.write("\nGlobal Summary Statistics\n")
    if all_sig:
        out.write(f"Signature Avg,{statistics.mean(all_sig)
:.3f}\n")
        out.write(f"Signature StdDev,{statistics.stdev(
all_sig):.3f}\n")
    if all_ver:
        out.write(f"Verification Avg,{statistics.mean(
all_ver):.3f}\n")
        out.write(f"Verification StdDev,{statistics.stdev(
all_ver):.3f}\n")

    # Per (W,H) summary
    out.write("\nPer-(W,H) Summary Statistics\n")
    out.write("W,H,Signature Avg,Signature StdDev,
Verification Avg,Verification StdDev\n")
    for (w,h) in sorted(per_wh_sig.keys()):
        sigs = per_wh_sig[(w,h)]
        vers = per_wh_ver[(w,h)]
        sig_avg = statistics.mean(sigs) if sigs else 0
        sig_std = statistics.stdev(sigs) if len(sigs) > 1
else 0
        ver_avg = statistics.mean(vers) if vers else 0
        ver_std = statistics.stdev(vers) if len(vers) > 1
else 0
        out.write(f"{w},{h},{sig_avg:.3f},{sig_std:.3f},{
ver_avg:.3f},{ver_std:.3f}\n")

print(f"Done! Results saved in {OUTPUT_FILE}")
```

Listing A.10: Python script for processing LMS benchmark logs and generating summary statistics.

A.6.1 Parsing and Visualizing gprof Results

To analyze profiling data from `gprof`, a custom Python script was written. It parses LaTeX tables produced from `gprof` outputs, excludes irrelevant functions (e.g., zeroization), and aggregates the execution time spent in SHA-256 hash routines versus auxiliary operations. The resulting statistics were used to generate stacked bar plots in Section 5.3.3.

```
import re
import matplotlib.pyplot as plt
import pandas as pd

# ------------------------------
# Configure
# ------------------------------
tex_file = "sw_gprof.tex"


EXCLUDE_FUNCS = {
    "hss_zeroize",
    "hss_seed_derive_set_j",
    "hss_seed_derive_set_q",
    "hss_finalize_hash_context",
    "hss_init_hash_context",
    "hss_combine_internal_nodes",
    "hss_compute_internal_node",
    "lm_ots_look_up_parameter_set",
    "hss_sign_update",
    "lm_ots_coef",
    "hss_validate_signature_update",
}

# ------------------------------
# Step 1: Parse LaTeX tables
# ------------------------------
with open(tex_file, "r") as f:
    tex = f.read()

# Regex to capture each table
tables = re.findall(
    r"\\caption\{Gprof flat profile for profile (.*?) \(h=20, w=(\d)\)\}.*?name \\\\\s*\\hline\s*(.*?)\\end\{tabular\}",
    tex,
    re.S
```

```
)


records = []

for profile, w, body in tables:
    # Extract lines like "44.14 & ... & hss_zeroize"
    row_re = re.compile(
    r"^\s*([\d\.]+)\s*&\s*([\d\.]+)\s*&\s*([\d\.]+)\s*&\s*(\d+)\s*&\s*([A-Za-z0-9_\\]+)\s*\\\\",
    re.M
    )
    rows = row_re.findall(body)

    print(f"--- Profile={profile}, w={w}, rows={len(rows)} ---")
    for r in rows[:3]:
        print(r)


    for pct, cum, selfsec, calls, name in rows:
        records.append({
            "profile": profile.strip(),
            "w": int(w),
            "function": name.replace("\\_", "_").strip(),
            "pct": float(pct),
            "calls": int(calls)
        })


df = pd.DataFrame(records)

# ------------------------------
# Step 2: Categorize functions
# ------------------------------

print("=== Unique functions in input ===")
print(df["function"].unique())
print("================================")

def is_hash_related(func_name: str) -> bool:
    return func_name in {
        "hss_seed_derive",
        "hss_hash",
        "hss_hash_ctx",
```

```
        "hss_update_hash_context",
        "lm_ots_generate_public_key"
    }


df["category"] = df["function"].apply(
    lambda x: "hash" if is_hash_related(x) else "other"
)

# Exclude non-algorithmic functions
df = df[~df["function"].isin(EXCLUDE_FUNCS)].copy()


# ------------------------------
# Step 3: Aggregate per profile and w
# ------------------------------
agg = df.groupby(["profile", "w", "category"])["pct"].sum().reset_index()
pivot = agg.pivot_table(index=["profile", "w"], columns="category", values="pct", fill_value=0).reset_index()

# Ensure both columns exist
if "hash" not in pivot.columns:
    pivot["hash"] = 0
if "other" not in pivot.columns:
    pivot["other"] = 0


# Renormalize to 100%
pivot["total"] = pivot["hash"] + pivot["other"]
pivot["hash_pct"] = 100 * pivot["hash"] / pivot["total"]
pivot["other_pct"] = 100 * pivot["other"] / pivot["total"]

# ------------------------------
# Step 4: Plot stacked bars
# ------------------------------
profiles = ["genkey", "sign", "verify"]
titles = {"genkey": "Keygen", "sign": "Sign", "verify": "Verify"}

for profile in profiles:
    data = pivot[pivot["profile"] == profile].sort_values("w")
    plt.figure(figsize=(6, 5))
```

```
    plt.bar(data["w"], data["hash_pct"], label="Hash", color
   ="tab:blue")
    plt.bar(data["w"], data["other_pct"], bottom=data["
   hash_pct"], label="Other", color="tab:orange")
    plt.xticks(data["w"])
    plt.xlabel("Winternitz parameter w")
    plt.ylabel("% time")
    plt.title(f"{titles[profile]} (h=20)")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"figure_{profile}.png", dpi=300)

plt.show()
```

Listing A.11: Python script for parsing `gprof` outputs and producing aggregated profiling statistics.

A.6.2 SHA-256 Cycle Counter Testbench

To measure the intrinsic latency of the SHA-256 core, the VHDL testbench was instrumented with a cycle counter that starts when `msg_valid` is asserted and stops when `hash_valid` is raised.

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
  use ieee.numeric_std.all;
  use work.sha256_pkg.all;

entity sha256_tb is
end sha256_tb;
architecture testbench of sha256_tb is
signal sha_msg_test : sha_array(1 downto 0) := (
  (x"6162636462636465636465666465666 7" &
   x"6566676866676869676869 6a68696a6b" &
   x"696a6b6c6a6b6c6d6b6c6d6e6c6d6e6f" &
   x"6d6e6f706e6f707180000000000000000"),
  (x"0000000000000000000000000000000 0" &
   x"0000000000000000000000000000000 0" &
   x"0000000000000000000000000000000 0" &
   x"0000000000000000000000000001c0")
);

signal sha_hash_test, sha_hash_func, sha_hash_func2,
   sha_hash_test2 : sha_hash;
```

```
22 signal clk, rst, msg_valid, hash_ack, hash_valid, mux_sel,
      word_sel,  msg_ready, en, rst_AE, en_DM, mux_sel_AE,
      en_DM_AE, msg_last, en_AE, msg_ready2, hash_valid2 :
      std_ulogic;
23 signal w_array : word_vector(15 downto 0);
24 signal w_array_exp : word_vector(63 downto 0);
25 signal mi, wi, mi1, mi2 : word;
26 signal mux_sel_H : std_ulogic_vector(1 downto 0);
27 signal K_index : natural range 0 to 64;
28 signal cycle_counter : integer := 0;
29 signal start_counting : boolean := false;
30 signal start_cycle : integer := 0;
31 signal end_cycle : integer := 0;
32
33
34 begin
35
36   sha_hash_func <= sha256(sha_msg_test, 2);
37   --sha_hash_func2 <= sha256(sha_msg_test2);
38   --w_array <= break_chunks(sha_msg_test);
39   --w_array_exp <= expand_msg_blocks(w_array);
40   clk_proc: process
41     begin
42       clk <= '1';
43       wait for 1 ns;
44       clk <= '0';
45       wait for 1 ns;
46   end process;
47
48   rst_proc: process
49     begin
50       rst<='0';
51       wait for 10 ns;
52       rst <= '1';
53       wait;
54   end process rst_proc;
55
56   exp_unit_test: process
57     begin
58       msg_last <= '0';
59       mi <= (others => '0');
60       wait for 30 ns;
61       for i in 0 to 15 loop
62         mi <= return_chunk(sha_msg_test(1), i);
63         wait for 2 ns;
```

```
64        end loop;
65        wait for 98 ns;
66        for i in 0 to 15 loop
67          mi <= return_chunk(sha_msg_test(0), i);
68          wait for 2 ns;
69        end loop;
70        msg_last <= '1';
71        wait;
72    end process;
73
74    cu_test : process
75      begin
76        msg_valid <= '0';
77        hash_ack <= '0';
78        wait for 30 ns;
79        msg_valid <= '1';
80        wait for 10 ns;
81        msg_valid <= '0';
82        wait for 10 ns;
83        msg_valid <= '1';
84        wait for 242 ns;
85        --msg_valid <= '0';
86        ---wait for 10 ns;
87        --msg_valid <= '1';
88        --wait for 142 ns;
89        msg_valid <= '0';
90        hash_ack <= '1';
91        wait;
92    end process cu_test;
93
94    --dut : entity work.sha256_exp_unit(arch) port map (clk,
      rst, mi, word_sel, wi);
95    --cu : entity work.sha256_cu(fsm) port map(clk, rst,
      msg_valid, msg_last, hash_ack, hash_valid, mux_sel,
      mux_sel_AE, word_sel, mux_sel_H, K_index, msg_ready, en,
      en_AE, en_DM, en_DM_AE, rst_AE);
96    --dp : entity work.sha256_core(arch) port map(clk, rst,
      rst_AE, mux_sel, mux_sel_AE, word_sel,  mux_sel_H,
      K_index, mi, en, en_AE , en_DM, en_DM_AE, sha_hash_test);
97    dut : entity work.sha256(struct) port map (clk, rst,
      msg_valid, msg_last, hash_ack, mi, msg_ready2,
      hash_valid2, sha_hash_test2);
98
99    cycle_counter_proc : process(clk)
100 begin
```

```
101     if rising_edge(clk) then
102         cycle_counter <= cycle_counter + 1;
103 
104         -- Start counting when msg_valid goes high
105         if msg_valid = '1' and not start_counting then
106             start_cycle <= cycle_counter;
107             start_counting <= true;
108         end if;
109 
110         -- Stop counting when hash_valid2 goes high
111         if hash_valid2 = '1' and start_counting then
112             end_cycle <= cycle_counter;
113             start_counting <= false;
114             report "SHA256 operation took " & integer'image(
    end_cycle - start_cycle) & " clock cycles";
115         end if;
116     end if;
117 end process;
118 
119 wait;
120 end testbench;
```

Listing A.12: Instrumented SHA-256 VHDL testbench with cycle counter.

A.6.3 C Microbenchmark for `perf`

To compare the cycle cost of software and hardware hashing in practice, a C microbenchmark was written. It calls either OpenSSL's SHA-256 or the hardware wrapper for a configurable number of iterations, while `perf stat` collects cycle counts.

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <stdint.h>
6  #include <sched.h>
7  #include "sha256_acc.h"
8  #include "openssl/sha.h"
9 
10 #define MAX_INPUT_LEN 1024
11 
12 #ifndef ITERATIONS
13 #define ITERATIONS 100000  // default if not specified at
    compile time
```

```c
#endif

int main(void) {

  cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(0, &set); // Use core 0

    if (sched_setaffinity(0, sizeof(set), &set) == -1) {
        perror("sched_setaffinity");
        return 1;
    }

    char input[MAX_INPUT_LEN] =
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
    size_t len = strlen(input);

#ifdef HARDWARE_HASH
    sha256_acc_init();
    uint32_t hash_result[8];

    for (int i = 0; i < ITERATIONS; i++) {
        hw_hash(input, len, hash_result);
    }

    sha256_acc_close();

#elif defined(SOFTWARE_HASH)
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256_CTX ctx;

    for (int i = 0; i < ITERATIONS; i++) {
        SHA256_Init(&ctx);
        SHA256_Update(&ctx, input, len);
        SHA256_Final(hash, &ctx);
    }
#endif

    return 0;
}
```

Listing A.13: C microbenchmark used with `perf` to measure cycle counts.

A.6.4 SHA-256 Cycle Benchmark Script

The following shell script automates the execution of the SHA-256 cycle microbenchmarks at different iteration counts, invoking `perf` and collecting results for later processing.

```
#!/bin/bash

# Define iteration counts
ITERATIONS_LIST=(1 10 100 1000)
ROUNDS=50

SRC="perf_sha256_test_final.c"
SHA_SRC="sha256_acc.c"
SHA_HDR="sha256_acc.h"

# Ensure script is run with sudo for perf
if [[ $EUID -ne 0 ]]; then
   echo "This script must be run with sudo."
   exit 1
fi

for ITER in "${ITERATIONS_LIST[@]}"; do
    for MODE in HW SW; do

        FOLDER="${ITER}IT_${MODE}"
        LOG_FILE="${FOLDER}/perf_cycles.log"
        EXE_FILE="${FOLDER}/hash_${MODE,,}_perf"

        echo ">>> Building ${MODE} for ITERATIONS=${ITER}"

        if [ "$MODE" == "HW" ]; then
            gcc -DHARDWARE_HASH -DITERATIONS=${ITER} "$SRC" "$SHA_SRC" -o "$EXE_FILE" -lcrypto
        else
            gcc -DSOFTWARE_HASH -DITERATIONS=${ITER} "$SRC" -o "$EXE_FILE" -lcrypto
        fi

        # Check build success
        if [ $? -ne 0 ]; then
            echo " Build failed for ${MODE} ITER=${ITER}"
            exit 1
        fi

        # Empty log file
```

```
        : > "$LOG_FILE"

        echo ">>> Running ${ROUNDS} perf rounds in ${FOLDER}..."
        for ((i=1; i<=ROUNDS; i++)); do
            CYCLES=$(sudo perf stat "$EXE_FILE" 2>&1 | grep cycles | awk '{gsub(",","",$1); print $1}')
            echo "$CYCLES" >> "$LOG_FILE"
        done

    done
done

echo "All tests completed."
```

Listing A.14: Shell script for automated SHA-256 cycle count benchmarking.

A.6.5 ECDSA Benchmarking Program and Script

Benchmarking of ECDSA P-256 signing and verification on the PYNQ-Z2 is performed in two stages:

- A **C benchmarking program** that performs key generation, signing, and verification on a fixed firmware image using OpenSSL. It prints signing and verification times for each operation.
- A **shell script** that repeatedly executes the program over multiple rounds, collecting timing data into log files and computing summary statistics (mean and standard deviation) using `awk`.

ECDSA C Benchmark Program

The C program uses OpenSSL to generate a key pair, perform a single signing and verification operation, and output the results in a machine-parsable format.

```
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define PRIV_KEY_FILE "ecdsa_priv.pem"
#define PUB_KEY_FILE  "ecdsa_pub.pem"
```

```
#define SIG_FILE      "signature.bin"

static void handle_openssl_error(void) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
}

static EVP_PKEY* ecdsa_keygen(void) {
    EVP_PKEY_CTX *pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC,
   NULL);
    if (!pctx) handle_openssl_error();
    if (EVP_PKEY_keygen_init(pctx) <= 0)
   handle_openssl_error();
    if (EVP_PKEY_CTX_set_ec_paramgen_curve_nid(pctx,
   NID_X9_62_prime256v1) <= 0)
        handle_openssl_error();

    EVP_PKEY *pkey = NULL;
    if (EVP_PKEY_keygen(pctx, &pkey) <= 0)
   handle_openssl_error();

    EVP_PKEY_CTX_free(pctx);
    return pkey;
}

static void save_private_key(EVP_PKEY *pkey, const char *
   filename) {
    FILE *fp = fopen(filename, "wb");
    if (!fp) { perror("Failed to open private key file");
   exit(EXIT_FAILURE); }
    if (!PEM_write_PrivateKey(fp, pkey, NULL, NULL, 0, NULL,
    NULL)) handle_openssl_error();
    fclose(fp);
}

static void save_public_key(EVP_PKEY *pkey, const char *
   filename) {
    FILE *fp = fopen(filename, "wb");
    if (!fp) { perror("Failed to open public key file");
   exit(EXIT_FAILURE); }
    if (!PEM_write_PUBKEY(fp, pkey)) handle_openssl_error();
    fclose(fp);
}

static EVP_PKEY* load_private_key(const char *filename) {
```

```
47      FILE *fp = fopen(filename, "rb");
48      if (!fp) { perror("Failed to open private key file");
    exit(EXIT_FAILURE); }
49      EVP_PKEY *pkey = PEM_read_PrivateKey(fp, NULL, NULL,
    NULL);
50      fclose(fp);
51      if (!pkey) handle_openssl_error();
52      return pkey;
53  }
54
55  static EVP_PKEY* load_public_key(const char *filename) {
56      FILE *fp = fopen(filename, "rb");
57      if (!fp) { perror("Failed to open public key file");
    exit(EXIT_FAILURE); }
58      EVP_PKEY *pkey = PEM_read_PUBKEY(fp, NULL, NULL, NULL);
59      fclose(fp);
60      if (!pkey) handle_openssl_error();
61      return pkey;
62  }
63
64  static unsigned char* read_file(const char *filename, size_t
      *len) {
65      FILE *fp = fopen(filename, "rb");
66      if (!fp) { perror("Failed to open file"); exit(
    EXIT_FAILURE); }
67      fseek(fp, 0, SEEK_END);
68      long size = ftell(fp);
69      rewind(fp);
70      unsigned char *buf = malloc(size);
71      if (!buf) { perror("malloc"); exit(EXIT_FAILURE); }
72      fread(buf, 1, size, fp);
73      fclose(fp);
74      *len = size;
75      return buf;
76  }
77
78  static void save_signature(const unsigned char *sig, size_t
    sig_len) {
79      FILE *fp = fopen(SIG_FILE, "wb");
80      if (!fp) { perror("Failed to open signature file"); exit
    (EXIT_FAILURE); }
81      fwrite(sig, 1, sig_len, fp);
82      fclose(fp);
83  }
84
```

```
85  int main(int argc, char *argv[]) {
86      OpenSSL_add_all_algorithms();
87      ERR_load_crypto_strings();
88
89      if (argc < 2) {
90          fprintf(stderr, "Usage:\n");
91          fprintf(stderr, "  %s keygen\n", argv[0]);
92          fprintf(stderr, "  %s signverify <file-to-sign>\n",
    argv[0]);
93          return EXIT_FAILURE;
94      }
95
96      if (strcmp(argv[1], "keygen") == 0) {
97          struct timespec t0, t1;
98          clock_gettime(CLOCK_MONOTONIC, &t0);
99          EVP_PKEY *pkey = ecdsa_keygen();
100         clock_gettime(CLOCK_MONOTONIC, &t1);
101
102         double elapsed_ms = (t1.tv_sec - t0.tv_sec) * 1000.0
    +
103                             (t1.tv_nsec - t0.tv_nsec) / 1e6;
104
105         save_private_key(pkey, PRIV_KEY_FILE);
106         save_public_key(pkey, PUB_KEY_FILE);
107         EVP_PKEY_free(pkey);
108
109         printf("[Keygen] Time: %.3f ms\n", elapsed_ms);
110     }
111
112     else if (strcmp(argv[1], "signverify") == 0) {
113         if (argc < 3) {
114             fprintf(stderr, "Missing file to sign.\n");
115             return EXIT_FAILURE;
116         }
117
118         // Load keys
119         EVP_PKEY *priv = load_private_key(PRIV_KEY_FILE);
120         EVP_PKEY *pub  = load_public_key(PUB_KEY_FILE);
121
122         // Read data
123         size_t data_len;
124         unsigned char *data = read_file(argv[2], &data_len);
125
126         // --- Sign ---
127         EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
```

```
128         if (!mdctx) handle_openssl_error();
129         if (EVP_DigestSignInit(mdctx, NULL, EVP_sha256(),
    NULL, priv) <= 0)
130             handle_openssl_error();
131 
132         size_t sig_len = 0;
133         EVP_DigestSign(mdctx, NULL, &sig_len, data, data_len
    );
134         unsigned char *sig = malloc(sig_len);
135         if (!sig) { perror("malloc"); exit(EXIT_FAILURE); }
136 
137         struct timespec ts0, ts1;
138         clock_gettime(CLOCK_MONOTONIC, &ts0);
139         EVP_DigestSign(mdctx, sig, &sig_len, data, data_len)
    ;
140         clock_gettime(CLOCK_MONOTONIC, &ts1);
141 
142         double sign_ms = (ts1.tv_sec - ts0.tv_sec) * 1000.0
    +
143                          (ts1.tv_nsec - ts0.tv_nsec) / 1e6;
144 
145         save_signature(sig, sig_len);
146         EVP_MD_CTX_free(mdctx);
147 
148         printf("[Sign] Time: %.3f ms, Signature size: %zu
    bytes\n", sign_ms, sig_len);
149 
150         // --- Verify ---
151         EVP_MD_CTX *mdctx_v = EVP_MD_CTX_new();
152         if (!mdctx_v) handle_openssl_error();
153         if (EVP_DigestVerifyInit(mdctx_v, NULL, EVP_sha256()
    , NULL, pub) <= 0)
154             handle_openssl_error();
155 
156         struct timespec tv0, tv1;
157         clock_gettime(CLOCK_MONOTONIC, &tv0);
158         int valid = EVP_DigestVerify(mdctx_v, sig, sig_len,
    data, data_len);
159         clock_gettime(CLOCK_MONOTONIC, &tv1);
160 
161         double verify_ms = (tv1.tv_sec - tv0.tv_sec) *
    1000.0 +
162                            (tv1.tv_nsec - tv0.tv_nsec) / 1e6
    ;
163 
```

```
164         if (valid == 1)
165             printf("[Verify] Time: %.3f ms (OK)\n",
    verify_ms);
166         else
167             fprintf(stderr, "[Verify] Failed or error\n");
168 
169         EVP_MD_CTX_free(mdctx_v);
170         EVP_PKEY_free(priv);
171         EVP_PKEY_free(pub);
172         free(data);
173         free(sig);
174     }
175 
176     else {
177         fprintf(stderr, "Unknown command '%s'\n", argv[1]);
178         return EXIT_FAILURE;
179     }
180 
181     EVP_cleanup();
182     ERR_free_strings();
183     return EXIT_SUCCESS;
184 }
```

Listing A.15: C program for benchmarking ECDSA P-256 signing and verification using OpenSSL.

ECDSA Shell Benchmark Script

The accompanying shell script automates the execution of the C benchmark over multiple rounds, logging all results for later processing.

```
1  #!/bin/bash
2 
3  # --------------------------------
4  # Configuration
5  # --------------------------------
6  ROUNDS=30
7  FIRMWARE_FILE="firmware.bin"
8  EXEC="./ecdsa_bench"
9  LOG_FILE="ecdsa_results.log"
10 
11 # --------------------------------
12 # Check prerequisites
13 # --------------------------------
14 if [ ! -f "$EXEC" ]; then
```

```
    echo "Error: $EXEC not found. Please compile ecdsa_bench.c first."
    exit 1
fi

if [ ! -f "$FIRMWARE_FILE" ]; then
    echo "Error: $FIRMWARE_FILE not found."
    exit 1
fi

# --------------------------------
# 1. Key Generation (once)
# --------------------------------
echo ">>> Generating ECDSA key pair..."
$EXEC keygen
if [ $? -ne 0 ]; then
    echo "Key generation failed."
    exit 1
fi

# --------------------------------
# 2. Benchmark sign + verify
# --------------------------------
echo ">>> Running $ROUNDS signing + verifying rounds..."
: > "$LOG_FILE"   # empty log file

for ((i=1; i<=ROUNDS; i++)); do
    echo "Round $i/$ROUNDS"
    # Capture the output of the program
    OUT=$($EXEC signverify "$FIRMWARE_FILE")

    # Extract numeric values
    SIGN_TIME=$(echo "$OUT" | grep "\[Sign\]" | awk '{print $3}')
    VERIFY_TIME=$(echo "$OUT" | grep "\[Verify\]" | awk '{print $3}')
    SIG_SIZE=$(echo "$OUT" | grep "\[Sign\]" | awk '{print $7}')

    # Append to log file
    echo "$SIGN_TIME $VERIFY_TIME $SIG_SIZE" >> "$LOG_FILE"
done

echo ">>> Benchmark complete. Raw results saved in $LOG_FILE."
```

```
55
56 # -------------------------------
57 # 3. Compute mean and stddev
58 # -------------------------------
59 echo ">>> Summary statistics:"
60 awk '
61 {
62     sign[NR]=$1; verify[NR]=$2; size[NR]=$3;
63     sign_sum+=$1; verify_sum+=$2; size_sum+=$3
64 }
65 END {
66     n=NR
67     sign_mean=sign_sum/n; verify_mean=verify_sum/n;
    size_mean=size_sum/n
68     for(i=1;i<=n;i++){
69         sign_var+=(sign[i]-sign_mean)^2
70         verify_var+=(verify[i]-verify_mean)^2
71         size_var+=(size[i]-size_mean)^2
72     }
73     sign_stddev=sqrt(sign_var/n)
74     verify_stddev=sqrt(verify_var/n)
75     size_stddev=sqrt(size_var/n)
76     printf "Sign time:    %.3f ms    %.3f ms\n", sign_mean,
    sign_stddev
77     printf "Verify time:  %.3f ms    %.3f ms\n", verify_mean
    , verify_stddev
78     printf "Sig. size:    %.1f bytes    %.1f\n", size_mean,
    size_stddev
79 }' "$LOG_FILE"
```

Listing A.16: Shell script for automating ECDSA P-256 signing and verification benchmarks.

Appendix B

Benchmark Data

B.1 Software-only Results

B.1.1 Execution Times

Table B.1 reports the execution times for the software-only LMS implementation across all (w, h) parameter sets.

Table B.1: Average and standard deviation of LMS signing and verification latency over 30 rounds (ms).

w	h	Signature Avg	Signature StdDev	Verification Avg	Verification StdDev
1	5	495.98	12.45	491.50	1.67
1	10	502.16	5.42	491.22	1.93
1	15	505.83	2.88	491.15	1.42
1	20	510.39	2.55	491.86	1.87
2	5	494.59	5.32	491.29	1.48
2	10	499.25	8.96	491.98	2.72
2	15	502.15	1.56	491.66	2.15
2	20	514.05	46.22	492.23	2.86
4	5	495.74	4.43	493.06	3.43
4	10	504.38	4.80	492.51	2.32
4	15	510.27	2.96	492.50	1.85
4	20	514.78	1.76	492.91	2.55
8	5	531.25	37.53	501.66	3.03
8	10	577.24	9.40	501.71	2.45
8	15	637.27	13.70	501.05	2.62
8	20	677.45	15.95	500.86	1.65

B.1.2 gprof Profiling

Detailed profiling outputs collected with `gprof` are shown below. These data form the basis for the profiling analysis presented in Section 5.3.3.

Table B.2: Gprof flat profile for profile genkey (h=20, w=1)

% time	cumulative seconds	self seconds	calls	name
44.14	14.35	14.35	177996224	hss_zeroize
13.47	18.73	4.38	91486128	hss_seed_derive
12.24	22.71	3.98	346641	lm_ots_generate_public_key
11.23	26.36	3.65	274814861	put_bigendian
6.27	28.40	2.04	85711531	hss_hash
4.61	31.54	1.50	91677199	hss_hash_ctx
2.86	32.47	0.93	88947820	hss_update_hash_context
0.06	32.49	0.02	340894	hss_seed_derive_set_j
0.03	32.50	0.01	335435	hss_finalize_hash_context
0.03	32.51	0.01	91	hss_compute_internal_node
0.00	32.51	0.00	356319	lm_ots_look_up_parameter_set
0.00	32.51	0.00	349512	hss_seed_derive_set_q
0.00	32.51	0.00	327120	hss_init_hash_context
0.00	32.51	0.00	255	hss_combine_internal_nodes

Table B.3: Gprof flat profile for profile sign (h=20, w=1)

% time	cumulative seconds	self seconds	calls	name
56.98	0.49	0.49	678363	hss_zeroize
10.47	0.58	0.09	353130	hss_seed_derive
9.30	0.66	0.08	333638	hss_hash
8.14	0.73	0.07	1068695	put_bigendian
4.65	0.82	0.04	352050	hss_hash_ctx
3.49	0.85	0.03	1342	lm_ots_generate_public_key
1.16	0.86	0.01	359589	hss_update_hash_context
0.00	0.86	0.00	16384	hss_sign_update
0.00	0.86	0.00	1406	lm_ots_look_up_parameter_set
0.00	0.86	0.00	1362	hss_seed_derive_set_q
0.00	0.86	0.00	1332	hss_seed_derive_set_j
0.00	0.86	0.00	1286	hss_init_hash_context
0.00	0.86	0.00	1261	hss_finalize_hash_context
0.00	0.86	0.00	265	lm_ots_coef
0.00	0.86	0.00	84	hss_combine_internal_nodes

Table B.4: Gprof flat profile for profile verify (h=20, w=1)

% time	cumulative seconds	self seconds	calls	name
0.00	0.01	0.00	16651	hss_update_hash_context
0.00	0.01	0.00	16384	hss_validate_signature_update
0.00	0.01	0.00	289	put_bigendian
0.00	0.01	0.00	265	lm_ots_coef
0.00	0.01	0.00	150	hss_hash_ctx
0.00	0.01	0.00	9	get_bigendian
0.00	0.01	0.00	2	hss_finalize_hash_context
0.00	0.01	0.00	2	hss_init_hash_context

Table B.5: Gprof flat profile for profile genkey (h=20, w=2)

% time	cumulative seconds	self seconds	calls	name
39.71	8.22	8.22	106290392	hss_zeroize
12.32	10.77	2.55	406754	lm_ots_generate_public_key
12.03	13.26	2.49	53394132	hss_seed_derive
10.19	15.37	2.11	161807329	put_bigendian
9.47	17.33	1.96	158559073	hss_hash_ctx
5.80	20.19	1.20	51382097	hss_hash
2.32	20.67	0.48	52762352	hss_update_hash_context
0.10	20.69	0.02	91	hss_compute_internal_node
0.05	20.70	0.01	391959	hss_init_hash_context
0.00	20.70	0.00	414549	lm_ots_look_up_parameter_set
0.00	20.70	0.00	406766	hss_seed_derive_set_q
0.00	20.70	0.00	400349	hss_seed_derive_set_j
0.00	20.70	0.00	395255	hss_finalize_hash_context
0.00	20.70	0.00	255	hss_combine_internal_nodes

Table B.6: Gprof flat profile for profile sign (h=20, w=2)

% time	cumulative seconds	self seconds	calls	name
50.00	0.19	0.19	395430	hss_zeroize
15.79	0.25	0.06	1535	lm_ots_generate_public_key
10.53	0.29	0.04	193145	hss_hash
7.89	0.32	0.03	199075	hss_seed_derive
5.26	0.37	0.02	212801	hss_update_hash_context
2.63	0.38	0.01	603768	put_bigendian
0.00	0.38	0.00	589163	hss_hash_ctx
0.00	0.38	0.00	16384	hss_sign_update
0.00	0.38	0.00	1566	lm_ots_look_up_parameter_set
0.00	0.38	0.00	1546	hss_seed_derive_set_q
0.00	0.38	0.00	1510	hss_seed_derive_set_j
0.00	0.38	0.00	1482	hss_finalize_hash_context
0.00	0.38	0.00	1470	hss_init_hash_context

Table B.7: Gprof flat profile for profile verify (h=20, w=2)

% time	cumulative seconds	self seconds	calls	name
0.00	0.00	0.00	16519	hss_update_hash_context
0.00	0.00	0.00	16384	hss_validate_signature_update
0.00	0.00	0.00	234	hss_hash_ctx
0.00	0.00	0.00	157	put_bigendian
0.00	0.00	0.00	133	lm_ots_coef
0.00	0.00	0.00	9	get_bigendian
0.00	0.00	0.00	2	hss_finalize_hash_context
0.00	0.00	0.00	2	hss_init_hash_context

Table B.8: Gprof flat profile for profile genkey (h=20, w=4)

% time	cumulative seconds	self seconds	calls	name
26.88	4.32	4.32	472970237	hss_hash_ctx
21.34	7.75	3.43	66112542	hss_zeroize
19.54	10.89	3.14	493965	lm_ots_generate_public_key
6.29	14.40	1.01	32921985	hss_seed_derive
5.23	15.24	0.84	100210851	put_bigendian
3.24	15.76	0.52	32613410	hss_hash
1.74	16.04	0.28	31222898	hss_update_hash_context
0.12	16.06	0.02	128	hss_compute_internal_node
0.06	16.07	0.01	495690	lm_ots_look_up_parameter_set
0.00	16.07	0.00	493708	hss_seed_derive_set_q
0.00	16.07	0.00	489560	hss_seed_derive_set_j
0.00	16.07	0.00	488947	hss_finalize_hash_context
0.00	16.07	0.00	484500	hss_init_hash_context

Table B.9: Gprof flat profile for profile sign (h=20, w=4)

% time	cumulative seconds	self seconds	calls	name
23.73	0.14	0.14	288384	hss_zeroize
20.34	0.26	0.12	2075115	hss_hash_ctx
20.34	0.38	0.12	2142	lm_ots_generate_public_key
5.08	0.53	0.03	437957	put_bigendian
5.08	0.56	0.03	143972	hss_seed_derive
5.08	0.59	0.03	142845	hss_hash
0.00	0.59	0.00	153571	hss_update_hash_context
0.00	0.59	0.00	16384	hss_sign_update
0.00	0.59	0.00	2164	lm_ots_look_up_parameter_set
0.00	0.59	0.00	2136	hss_finalize_hash_context
0.00	0.59	0.00	2134	hss_seed_derive_set_q
0.00	0.59	0.00	2118	hss_seed_derive_set_j
0.00	0.59	0.00	2111	hss_init_hash_context

Table B.10: Gprof flat profile for profile verify (h=20, w=4)

% time	cumulative seconds	self seconds	calls	name
0.00	0.00	0.00	16453	hss_update_hash_context
0.00	0.00	0.00	16384	hss_validate_signature_update
0.00	0.00	0.00	531	hss_hash_ctx

Table B.11: Gprof flat profile for profile genkey (h=20, w=8)

% time	cumulative seconds	self seconds	calls	name
46.49	37.65	37.65	4084867923	hss_hash_ctx
27.35	59.80	22.15	567171	lm_ots_generate_public_key
1.94	79.49	1.57	39418036	hss_zeroize
0.64	80.01	0.52	19383016	hss_seed_derive
0.58	80.48	0.47	58928012	put_bigendian
0.46	80.85	0.37	19507588	hss_hash
0.10	80.93	0.08	15275320	hss_update_hash_context
0.06	80.98	0.05	122	hss_compute_internal_node
0.01	80.99	0.01	570412	lm_ots_look_up_parameter_set
0.00	80.99	0.00	574165	hss_seed_derive_set_q
0.00	80.99	0.00	571759	hss_seed_derive_set_j
0.00	80.99	0.00	566932	hss_finalize_hash_context
0.00	80.99	0.00	554688	hss_init_hash_context

Table B.12: Gprof flat profile for profile sign (h=20, w=8)

% time	cumulative seconds	self seconds	calls	name
46.03	0.29	0.29	16122520	hss_hash_ctx
36.51	0.52	0.23	2354	lm_ots_generate_public_key
1.59	0.63	0.01	80727	hss_seed_derive
0.00	0.63	0.00	243598	put_bigendian
0.00	0.63	0.00	164107	hss_zeroize
0.00	0.63	0.00	81220	hss_hash
0.00	0.63	0.00	74342	hss_update_hash_context
0.00	0.63	0.00	16384	hss_sign_update
0.00	0.63	0.00	2392	hss_seed_derive_set_q
0.00	0.63	0.00	2368	lm_ots_look_up_parameter_set
0.00	0.63	0.00	2362	hss_finalize_hash_context
0.00	0.63	0.00	2362	hss_seed_derive_set_j
0.00	0.63	0.00	2290	hss_init_hash_context

Table B.13: Gprof flat profile for profile verify (h=20, w=8)

% time	cumulative seconds	self seconds	calls	name
0.00	0.00	0.00	16420	hss_update_hash_context
0.00	0.00	0.00	16384	hss_validate_signature_update
0.00	0.00	0.00	4101	hss_hash_ctx

B.2 Hardware-accelerated Results

This section reports raw measurement data for the hardware-accelerated LMS implementation. Results are divided between the baseline (non- optimized) driver and the optimized version with `mmap()` access.

B.2.1 Non-Optimized Driver Execution Times

Tables B.14 and B.15 show signature generation and verification latencies before driver optimization, across all parameter sets.

Table B.14: Hardware accelerated signature generation times (ms) on PYNQ-Z2 with non-optimized memory access.

h	$w = 1$	$w = 2$	$w = 4$	$w = 8$
5	1880.469	1913.687	2012.551	3371.347
10	2005.605	1993.180	2165.307	4625.536
15	2122.998	2113.278	2404.307	6546.042
20	2201.208	2190.628	4024.542	7784.278
25	`out_of_memory`			

Table B.15: Hardware accelerated signature verification times (ms) on PYNQ-Z2 with non-optimized memory access.

h	$w = 1$	$w = 2$	$w = 4$	$w = 8$
5	1834.907	1843.802	1859.331	2153.720
10	1832.112	1841.410	1863.687	2156.418
15	1834.351	1841.437	1863.971	2124.255
20	1835.898	1843.445	1885.630	2141.855
25	`out_of_memory`			

B.2.2 Optimized Driver Execution Times

Tables B.16 and B.17 report results after driver optimization. Each configuration was repeated over 30 rounds; the tables show mean latencies and standard deviations. These data were used to compute the speedups reported in Section 5.4.2.

Table B.16: Hardware accelerated signature generation times (ms) on PYNQ-Z2 after driver optimization (30 rounds).

h	$w = 1$	$w = 2$	$w = 4$	$w = 8$
5	1847.387 ± 2.156	1858.488 ± 3.006	1885.732 ± 2.057	2320.352 ± 3.973
10	1892.182 ± 23.684	1883.553 ± 1.997	1936.780 ± 2.202	2689.925 ± 2.143
15	1927.240 ± 1.711	1921.105 ± 2.461	2006.931 ± 3.106	3272.497 ± 2.194
20	1954.115 ± 2.466	1945.469 ± 2.194	2055.915 ± 2.189	3661.065 ± 2.814

Table B.17: Hardware accelerated signature verification times (ms) on PYNQ-Z2 after driver optimization (30 rounds).

h	$w = 1$	$w = 2$	$w = 4$	$w = 8$
5	1834.907 ± 3.793	1835.413 ± 5.759	1843.893 ± 4.255	1921.594 ± 3.726
10	1879.940 ± 242.526	1837.515 ± 4.156	1842.254 ± 4.881	1944.156 ± 5.306
15	1835.565 ± 4.000	1836.131 ± 4.398	1842.550 ± 3.577	1948.075 ± 23.443
20	1846.462 ± 57.586	1836.977 ± 3.888	1843.526 ± 4.313	1944.800 ± 3.599

Bibliography

[1] David McGrew, Michael Curcio, and Scott Fluhrer. *Leighton-Micali Hash-Based Signatures*. RFC 8554. Apr. 2019. DOI: 10.17487/RFC8554. URL: https://www.rfc-editor.org/info/rfc8554 (cit. on pp. 5, 27, 28, 48).

[2] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. May 2018. DOI: 10.17487/RFC8391. URL: https://www.rfc-editor.org/info/rfc8391 (cit. on p. 5).

[3] Bas Westerbaan and Luke Valenta. *A look at the latest post-quantum signature standardization candidates*. URL: https://blog.cloudflare.com/another-look-at-pq-signatures/ (cit. on p. 5).

[4] *Signature zoo*. URL: https://pqshield.github.io/nist-sigs-zoo/ (cit. on p. 6).

[5] Alexander Wagner, Felix Oberhansl, and Marc Schink. «Extended version – to be or not to be stateful: post-quantum secure boot using hash-based signatures». In: *Journal of Cryptographic Engineering* (2024) (cit. on p. 9).

[6] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. *Trapdoors for Hard Lattices and New Cryptographic Constructions*. Cryptology ePrint Archive, Paper 2007/432. 2007. URL: https://eprint.iacr.org/2007/432 (cit. on p. 15).

[7] URL: https://csrc.nist.gov/CSRC/media/Presentations/Falcon/images-media/Falcon-April2018.pdf (cit. on p. 18).

[8] URL: https://falcon-sign.info/ (cit. on p. 18).

[9] Vadim Lyubashevsky. «Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures». In: *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*. Vol. 5912. Lecture Notes in Computer Science. Springer, 2009, pp. 598–616. DOI: 10.1007/978-3-642-10366-7_35. URL: https://www.iacr.org/archive/asiacrypt2009/59120596/59120596.pdf (cit. on p. 19).

[10] URL: https://pq-crystals.org/dilithium/ (cit. on p. 21).

[11] National Security Agency. *Commercial National Security Algorithm Suite 2.0*. Tech. rep. U.S. National Security Agency, Sept. 2022. URL: https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/0/CSI_CNSA_2.0_FAQ_.PDF (cit. on p. 22).

[12] D. Cooper, D. Apon, Q. Dang, M. Davidson, M. Dworkin, and C. Miller. *Recommendation for Stateful Hash-Based Signature Schemes*. Tech. rep. NIST Special Publication 800-208. Gaithersburg, MD: National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.SP.800-208. URL: https://doi.org/10.6028/NIST.SP.800-208 (cit. on p. 35).

[13] Martina Fogliato. *SHA256 Hardware Accelerator for Zybo Board*. https://github.com/martinafogliato/Sha256_Hw_Accelerator. 2020 (cit. on pp. 42, 44).

[14] Xilinx. *Xilinx Linux kernel (linux-xlnx)*. https://github.com/Xilinx/linux-xlnx. 2025 (cit. on p. 44).

[15] Cisco Systems. *hash-sigs: Hash-Based Signatures Reference Implementation*. https://github.com/cisco/hash-sigs/tree/master. 2025 (cit. on pp. 48, 49).

[16] *OpenSSL Project* (cit. on p. 48).

[17] Free Software Foundation. *The GNU C Library: Time Functions*. 2023. URL: https://www.gnu.org/software/libc/manual/html_node/Time-Functions.html (cit. on p. 53).

[18] GNU Project. *GNU gprof: The GNU Profiler*. 2023. URL: https://sourceware.org/binutils/docs/gprof/ (cit. on p. 53).

[19] Linux Kernel Organization. *Linux perf: Performance Counters for Linux*. 2023. URL: https://perf.wiki.kernel.org/index.php/Main_Page (cit. on p. 54).