POLITECNICO DI TORINO

Master's Degree in Computer Engineering



Master's Degree Thesis

Post-Quantum Secure Boot on the Infineon AURIX TC375TP

Supervisors

Prof. Alessandro SAVINO

Dr. Franco OBERTI

Prof. Stefano DI CARLO

Candidate

Luca CARPINTERI

OCTOBER 2026

Alla mia famiglia,
ai miei amici,
a tutti coloro che mi hanno sostenuto.

Abstract

A modern vehicle runs its functions on several Electronic Control Units, and secure boot checks the authenticity of the firmware each one runs through a digital signature. The signatures deployed today use RSA and elliptic-curve schemes, and a Cryptographically Relevant Quantum Computer would break both. A vehicle stays in service for more than a decade, so the migration to Post-Quantum Cryptography must begin before such a machine exists. The obstacle is that this cryptography has to run on hardware already in the field, and this hardware provides no acceleration: on the target platform the on-chip Hardware Security Module accelerates only classical primitives such as AES, SHA-2, and elliptic curve cryptography. This thesis investigates whether post quantum firmware verification fits the cycle, memory, and code-size budget of a secure bootloader on such a device, where verification is a software only task. The bootloader is implemented on the Infineon AURIX TC375TP. It authenticates firmware at boot, with dual-slot application partitioning, a signed metadata header that carries the public key and the signature, and a firmware-upload channel built on the lwIP network stack. Three lattice based post quantum and two classical schemes were implemented and benchmarked on the target with multiple parameter sets: ML-DSA with three security levels, Falcon and HAWK in both variants, Ed25519, and RSA-PSS with 2048 and 3072 bits. The forward and inverse Number Theoretic Transform (NTT) of ML-DSA were rewritten in TriCore assembly, following the ARM Cortex-M4 pqm4 twiddle-caching strategy remapped to the TriCore address registers. Every configuration was measured on the target silicon under one methodology: cache disabled, temporaries in Data Scratch-Pad RAM, a 64 KiB firmware image, and the message hash timed apart from the signature operation. The assembly NTT reduces the isolated NTT runtime by 94.1 % and the ML-DSA cryptographic verification by about 16 %. On the cryptographic verification phase the post-quantum schemes cost between 2.59 ms for Falcon-512 and 44.77 ms for ML-DSA-87. At the classical 128-bit strength of Ed25519 (19.28 ms), which corresponds to NIST level 1, every post quantum scheme at that level verifies faster than Ed25519. The memory cost stays inside the platform budget, with a worst-case verification stack of 91.0 KiB against a 125 KiB allocation, and a largest public-key and signature pair of 7,219 bytes inside the 8 KiB header. The message hash accounts for more than half of the verification time for every SHAKE-256 based scheme, and for 95.5 % of it for Falcon-512, and it grows linearly with the firmware size. On the cryptographic budget the migration is feasible: every post-quantum scheme runs in software without dedicated hardware and stays competitive with the classical baselines. On the end-to-end boot budget no scheme is feasible, and this does not follow from the migration. The binding

cost is the firmware hash, which already exceeds 50 ms for a 64 KiB image making a hardware hash accelerator mandatory on a conventional bootloader.

Table of Contents

List of Tables v

List of Figures vii

1 Introduction 1
- 1.1 Motivation and Threats . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Objectives and Scope . . . 3
- 1.4 Related Work . . . 4
- 1.5 Contributions . . . 5
- 1.6 Thesis Outline . . . 6

2 Post Quantum Cryptography 7
- 2.1 Quantum Computational Threats to Cryptographic Architectures . 8
  - 2.1.1 Shor's Algorithm and Public-Key Cryptography . . . 8
  - 2.1.2 Grover's Algorithm and Symmetric Security . . . 8
  - 2.1.3 The Brassard-Høyer-Tapp Algorithm . . . 9
  - 2.1.4 NIST PQC Standardization . . . 9
  - 2.1.5 NIST PQC Security Levels . . . 10
- 2.2 Selection of Evaluated Schemes . . . 11
- 2.3 ML-DSA . . . 12
  - 2.3.1 Number Theoretic Transform . . . 13
  - 2.3.2 ML-DSA Signature Protocol . . . 14
  - 2.3.3 ML-DSA Operations . . . 16
- 2.4 Falcon . . . 21
  - 2.4.1 GPV framework . . . 21
  - 2.4.2 NTRU lattices . . . 21
  - 2.4.3 Underlying hard problems . . . 22
  - 2.4.4 Falcon Operations . . . 22
- 2.5 HAWK . . . 29
  - 2.5.1 Key structure and trapdoor . . . 29

2.5.2 Security assumptions . . . 29
2.5.3 Comparison with Falcon . . . 30
2.5.4 HAWK operations . . . 30

3 Bootloader Rationale and Implementation 38
3.1 Bootloader Fundamentals . . . 38
3.1.1 Bootloader Rationale . . . 39
3.1.2 Structure and Execution Flow . . . 39
3.1.3 Secure Bootloader . . . 41
3.1.4 Bootloader Design Classes . . . 41
3.2 AURIX Development Board . . . 42
3.2.1 Hardware Security Module . . . 43
3.2.2 Development Board and Environment . . . 44
3.3 Bootloader Architecture and Design . . . 46
3.3.1 System Architecture . . . 46
3.3.2 Memory Layout and Slot Organization . . . 48
3.3.3 AURIX Application Binary Code Organization . . . 50
3.3.4 Threat Model . . . 50
3.3.5 Application Metadata Header . . . 52
3.3.6 Boot Decision Algorithm . . . 56
3.3.7 Cryptographic Verification . . . 62
3.3.8 Optimized NTT Implementation . . . 65
3.3.9 Multi-Core Application Startup . . . 67
3.3.10 Update Subsystem . . . 68
3.3.11 Signing Tool and Trust Anchor Provisioning . . . 69

4 Benchmark 71
4.1 Benchmark Setup . . . 71
4.1.1 Measurement Methodology . . . 71
4.1.2 Test Message . . . 72
4.1.3 Hash Libraries . . . 72
4.1.4 Algorithm Configurations . . . 73
4.1.5 Memory Metrics . . . 73
4.2 Hash Baseline . . . 73
4.3 Key Generation . . . 75
4.4 Signature Generation . . . 77
4.5 Signature Verification . . . 83
4.6 NTT Optimization Impact . . . 87
4.7 ROM Footprint . . . 87
4.8 Discussion . . . 90
4.8.1 Hash Dominance . . . 90

4.8.2 Comparison at Matched Security Levels . . . 90
4.8.3 Algorithm Selection for the Bootloader . . . 91
4.8.4 Comparison with Other Work . . . 91

5 Conclusion and Future Work 94
5.1 Summary of the Work . . . 94
5.2 Principal Findings . . . 95
5.2.1 Feasibility of Post-Quantum Verification on the Target . . . 95
5.2.2 Hash Latency Dominance . . . 96
5.2.3 Limited System-Level Impact of NTT . . . 96
5.2.4 Memory Footprint Fits Embedded Deployment . . . 97
5.2.5 Suitability of the Evaluated Schemes . . . 97
5.2.6 Cost of Post-Quantum Migration . . . 97
5.3 Limitations . . . 98
5.4 Future Work . . . 99
5.4.1 Reducing the Dominant Hash Cost . . . 100
5.4.2 Parallel Verification and Deferred Enforcement . . . 100
5.4.3 Scheme-Specific Transform Optimization . . . 101
5.4.4 Key Management and the Update Channel . . . 101
5.5 Closing Remarks . . . 102

A Clock Cycle Benchmarks 103

B Signer 109
B.1 Firmware Signer Script (`sign.py`) . . . 109
B.2 Python C-Extension Wrapper (`pyqcsign.pyd.c`) . . . 116

C AURIX Benchmark Harness 120
C.1 CPU Performance Counter Harness . . . 120

D AURIX Assembly NTT Implementation 121
D.1 Forward NTT Assembly Routine . . . 121
D.2 Inverse NTT Assembly Routine . . . 124
D.3 NTT Assembly Utilities and Macros . . . 129

Bibliography 133

List of Tables

2.1 Classical security strength and the corresponding RSA modulus and elliptic-curve key sizes, based on NIST SP 800-57 Part 1 Revision 5 [25]. 8

2.2 Computational complexity of classical vs. quantum attacks on symmetric and hash primitives. 9

2.3 ML-DSA parameter sets. The modulus $q = 8\,380\,417$, the degree $n = 256$, the root of unity $\zeta = 1753$, and the dropped-bit count $d = 13$ are identical across the three sets. `try` is the mean number of signing attempts. 12

3.1 TriCore register map and context classification. 45

3.2 Bootloader and application regions in program flash (non-cached segment). 49

3.3 Metadata header field layout (96 bytes, little-endian). 54

3.4 Macro Header Memory Layout (8192 Bytes Total). 56

3.5 Boot Candidate Selection Decision Table 61

3.6 Supported signature algorithms. PQC indicates a post-quantum scheme. Key and signature sizes are in bytes. 63

3.7 Comparative architectural resource utilization for the 3-3-2 merged NTT. 66

3.8 ML-DSA NTT and INTT cycle counts on the TC375TP microcontroller. 67

4.1 Mean hash time for the 65,536-byte test message by hash function. 74

4.2 Key generation statistics per algorithm variant. Times in milliseconds at 300 MHz; key sizes in bytes; worst-case stack depth in KiB (Section 4.1.5). 76

4.3 Signature generation statistics. Times in milliseconds at 300 MHz; signature size in bytes; worst-case stack depth in KiB (Section 4.1.5). sign_σ and $\text{sign}_{\max}$ refer to the crypto phase only. Tries (max) reports mean signing attempts with observed maximum in parentheses. ⋆ RSA signing is constant to within <1 μs across all iterations. . . . 78
4.4 Signature verification statistics. Times in milliseconds at 300 MHz; header material in bytes; worst-case stack depth and ROM in KiB (Section 4.1.5). Hash% is the fraction of total verification time spent in message hashing. pk+sig is the sum of the public key and signature sizes. 84
4.5 ML-DSA reference vs. assembly NTT: mean crypto timings. All times in milliseconds. Delta% is the relative improvement of the asm variant over the reference. 87
4.6 ROM footprint of cryptographic library object files only. 88
4.7 Cryptographic verification time on the TriCore 1.6.2 at 300 MHz: this work on the TC375TP versus Oka and Jungklass on the TC397XX [14]. Times in milliseconds. Oka and Jungklass report two memory configurations (see text). Their labels ML-DSA 2/3/5 correspond to ML-DSA-44/65/87, and FN-DSA to Falcon. 92

A.1 Key generation per algorithm variant, in million clock cycles (Mcc means 10^6 cycles; at 300 MHz 1 Mcc $\approx$ 3.33 ms). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072). *OVF* marks a counter overflow ($> 2^{31} - 1$ cycles, $\approx$ 7.16 s). 104
A.2 Message hashing (signing phase, 64 KiB message) per algorithm variant, in raw clock cycles (cc; at 300 MHz 1 cycle $\approx$ 3.33 ns). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072). . 105
A.3 Signature generation (cryptographic phase) per algorithm variant, in million clock cycles (Mcc means 10^6 cycles; at 300 MHz 1 Mcc $\approx$ 3.33 ms). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072). 106
A.4 Message hashing (verification phase, 64 KiB message) per algorithm variant, in raw clock cycles (cc; at 300 MHz 1 cycle $\approx$ 3.33 ns). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072). . 107
A.5 Signature verification (cryptographic phase) per algorithm variant, in raw clock cycles (cc; at 300 MHz 1 cycle $\approx$ 3.33 ns). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072). 108

List of Figures

2.1 Two-dimensional toy representation of the Falcon signature scheme geometry (not to scale). 28
2.2 Two-dimensional toy representation of the HAWK signature scheme geometry. 37

3.1 Boot stages on a microcontroller. The BootROM is sealed at manufacturing time and cannot be corrected in the field. The bootloader is the first stage implemented in software. A single software stage is sufficient when the application image is executed in place from flash memory. 40
3.2 Bootloader system architecture. The ROM bootloader (first stage, immutable) transfers execution to the custom second-stage bootloader. The second-stage bootloader authenticates a firmware candidate in Slot A or Slot B and launches the application; if no valid candidate is found, the lwIP-based update subsystem activates to receive a new image via Ethernet into Slot B. The host-side signing tool `sign.py` produces the firmware image transmitted over TCP. The boot decision algorithm is described in Section 3.3.6. 49
3.3 Memory layout of an AURIX TC375TP firmware slot (Slot A, base address `0xA0300000`). The 8 KiB metadata header precedes the application binary, which begins at `0xA0302000` where the first INTVEC lands on a natural 8 KiB-aligned boundary. *Approximate start of `.text`. 53
3.4 Bootloader Metadata Header Layout (8 Bytes per Row) 55
3.5 Macro Header High-Level Memory Map. 57

4.1 Key generation time for all variants (mean ± standard deviation, log scale). ML-DSA is quasi-deterministic (error bars invisible at this scale); Falcon and HAWK show large variance from NTRU-solve restarts. RSA overflows the ≈7160 ms counter limit for all four variants. 77

4.2 Mean signature generation time: hash phase (left bar) and cryptographic phase (right bar), log scale. ML-DSA assembly build only; RSA-2048 and RSA-3072 each shown once (timing is identical for $e = 3$ and $e = 65537$). Labels show hash time and sign time in ms. . 79
4.3 Distribution of signing attempts (`sign_try`) per ML-DSA parameter set, 1000 iterations each. A value of 1 means success on the first attempt. The ref and asm builds produce statistically identical distributions for each level (same DRBG seed sequence). 81
4.4 ML-DSA signature generation time: C reference vs. assembly NTT, all three parameter sets. Hash phase (lower) and crypto phase (upper). Labels show $\text{sign}_{\text{mean}}$ and sign_{P99} in ms. 82
4.5 Mean signature verification time: hash phase (left bar) and cryptographic phase (right bar), linear scale. ML-DSA assembly build only. All RSA variants are omitted for scale. Labels show hash time and crypto time in ms. 85
4.6 ML-DSA signature verification time: C reference vs. assembly NTT, all three parameter sets. Hash phase (lower) and crypto phase (upper). 86
4.7 ROM footprint of cryptographic library object files per algorithm, split into `.text` (executable code, solid) and `.rodata` (read-only constants and lookup tables, lighter shade). 89

Chapter 1

Introduction

Modern road vehicles integrate up to 100 Electronic Control Units (ECUs) [1]. An ECU is an embedded device built around a Microcontroller Unit (MCU) that integrates a processor core, volatile and non-volatile memory, and input/output peripherals. ECUs process sensor data in real time and exchange messages over the in-vehicle network to control vehicular subsystems. These subsystems include the powertrain, the anti-lock braking system, battery and thermal management in electrified vehicles, and advanced driver-assistance systems (ADAS).

Because an ECU is a programmable device that executes software, it can be a target for malicious actions. Interconnected ECUs executing large codebases expand the vehicle's attack surface. External connectivity increases this vulnerability. Over-the-air (OTA) updates and Vehicle-to-Everything (V2X) communication expose the in-vehicle network to remote threats. An adversary who can alter the firmware of a single ECU can interfere with the systems it controls or pivot to other units on the shared vehicular network, such as the Controller Area Network (CAN). When the affected ECU drives a safety function, firmware tampering raises the risk of physical harm to vehicle occupants.

These safety risks require a verification mechanism to ensure the execution of authorized firmware. This mechanism is secure boot. Secure boot verifies the integrity and authenticity of the firmware image before execution. This verification relies on cryptographic operations performed over structured metadata. If the platform supports in-field updates, this mechanism also validates the incoming firmware image before installation [2].

Secure boot establishes a chain of trust. Execution begins in an immutable, factory-programmed boot ROM that reads and verifies the boot configuration before transferring control to the second-stage bootloader. The second-stage bootloader then authenticates the application firmware before transferring execution to it [3].

In the automotive domain, secure boot is a regulated requirement rather than an optional feature. United Nations Regulation No. 155 makes a certified Cybersecurity

Management System a precondition for vehicle type approval, and it has been mandatory for all new vehicles in several major markets since July 2024 [4].

1.1 Motivation and Threats

Secure boot in current vehicles relies on classical asymmetric cryptography. The digital signatures that authenticate firmware are built on RSA or elliptic-curve schemes [5], whose security depends on the intractability of the Integer Factorization Problem (IFP) and the Discrete Logarithm Problem (DLP). Shor's algorithm solves both the IFP and the DLP, including its elliptic-curve variant, in polynomial time on a sufficiently large quantum computer [6], invalidating these security assumptions. A Cryptographically Relevant Quantum Computer (CRQC) would therefore make RSA, the Digital Signature Algorithm, and Elliptic Curve Digital Signature Algorithm (ECDSA) schemes insecure. Chapter 2 analyzes the quantum threat and the post-quantum response in detail; this section states only the consequence for the automotive context.

The migration to Post-Quantum Cryptography (PQC) for vehicles must begin as soon as possible, the deadline is set by lead time, not by an active attack. Mosca's inequality formalizes this [7]: let x be the time an asset must remain secure, y the time required to migrate to quantum-resistant cryptography, and z the time until a CRQC becomes available; if $x + y > z$, the migration begins too late. For automotive platforms the migration time y spans an entire vehicle fleet whose service life exceeds a decade, so y stays large even when z is distant. Conservative estimates place a CRQC a decade or more away [8], yet that lead time still forces the transition to start now.

The target Infineon AURIX TC3xx platform provides no hardware acceleration for the Keccak permutation used by all the post-quantum schemes evaluated here, and its on-chip HSM lacks dedicated post-quantum engines and is usable only through a proprietary toolchain [9, 10, 11]. Post-quantum verification on this device is therefore a software-only problem; Chapter 3 details the HSM constraints.

The secure bootloader must complete verification within a tight time budget. This budget depends on the ECU's functional domain. For Controller Area Network (CAN) communication, the start-up window to begin handling network traffic ranges from 30 to 100 milliseconds [12, 13]. Controllers that implement safety or hard real-time functions are held to a tighter target of about 10 milliseconds from power-on, while controllers used for less critical functions may tolerate a longer initialization. More generally, automotive secure boot faces stricter real-time constraints than other embedded domains: safety controllers must become operational within milliseconds of ignition, so signature verification must complete without delaying vehicle operation [14].

Signature verification adds directly to this start-up path. Therefore, execution latency and on-chip memory consumption are the primary constraints for secure boot [15].

1.2 Problem Statement

It remains an open engineering problem whether standardized post-quantum signature verification satisfies the cycle-count, memory, and code-size constraints of a bare-metal secure bootloader executing on an automotive microcontroller. This thesis addresses this problem by implementing a post-quantum secure bootloader on the AURIX TC3xx architecture and measuring the verification overhead on board.

1.3 Objectives and Scope

The primary objective of this thesis is to implement and evaluate post-quantum signature verification for secure boot on the Infineon AURIX TC375TP microcontroller. This objective consists of three specific goals:

- design and implement a bare-metal second-stage secure bootloader for the target platform, including firmware update and verification logic;
- integrate classical and post-quantum signature schemes behind a single verification interface, and port the ML-DSA Number Theoretic Transform, to TriCore assembly;
- benchmark every scheme on the target and compare them in terms of execution time, memory footprint, and signature size, to characterize the feasibility of post-quantum secure boot on this family of device.

The evaluation covers twelve signature configurations: ML-DSA-44, ML-DSA-65, and ML-DSA-87; Falcon-512 and Falcon-1024; HAWK-512 and HAWK-1024; the classical baseline Ed25519; and RSA-PSS at 2048 and 3072 bits with public exponents $e = 3$ and $e = 65537$. ML-DSA is additionally evaluated in a portable C reference variant and in the optimized TriCore assembly variant, so that the effect of the optimization is measurable directly. Signing and key generation are evaluated for completeness, but the bootloader executes only verification on the device; signing is performed offline on a host. The stateful hash-based signatures LMS and XMSS, recommended for firmware signing by NIST SP 800-208 [16] and CNSA 2.0 [17], are outside the evaluated set; Section 5.3 discusses this as a limitation.

This work evaluates the performance and resource feasibility of post-quantum verification. The hardening expected of a production deployment lies outside that objective, and the boundaries below follow from it. Key storage uses a software-embedded trust anchor rather than HSM or one-time-programmable memory, so it is not protected against a physical attacker; Section 5.3 describes the production alternative. No side-channel or fault-injection attacks were mounted against the device, and their offensive evaluation is out of scope. The integrated third-party cryptographic libraries are not hardened against such attacks, while the bootloader's own boot decision logic carries software fault-injection countermeasures (Section 3.3.6). The code follows MISRA-C guidance only in part, and the integrated third-party libraries were not audited for compliance. Additional candidate schemes were evaluated and excluded before benchmarking: MAYO for its large memory and material expansion, SPHINCS+ for signatures that exceed the 8 KiB signed-header budget at all but the smallest parameter set [18], CROSS and LESS for large signature or key sizes, and SQIsign for an unstable reference implementation on the target.

1.4 Related Work

In the automotive domain, secure boot authenticates each stage with a digital signature and aborts execution on a failed check, which separates it from authenticated boot, where a failure is only recorded [1]. The chain of trust originates in a hardware root of trust, and extends through the bootloader to the application [19]. This thesis implements a stage of secure boot of this kind, rooted in software, with the trust anchor held in the bootloader image rather than in dedicated hardware (Section 3.1.4).

Post-quantum cryptography has already been measured on automotive oriented microcontrollers, but mostly outside the signature-based secure-boot path. Fritzmann et al. evaluated key-encapsulation mechanisms on the AURIX TC2xx family, and established the scratch-pad measurement methodology with disabled caches that this work also adopts [20, 21]. Ravi et al. benchmarked post-quantum authentication over the automotive CAN bus, and found that communication bandwidth, rather than local computation, bounds the protocol [22]. These studies target key exchange or in-vehicle communication, and do not evaluate firmware signature verification at boot.

A smaller part of work applies post-quantum signatures to secure boot directly. Kumar et al. implemented a hardware-accelerated bootloader on an FPGA using the stateful hash-based scheme XMSS, and reported verification in the low-millisecond range against a start-up budget of 30 to 100 milliseconds for CAN traffic [12]. Bos et al. implemented a CRYSTALS-Dilithium secure boot on the NXP S32G274A, a

high-performance vehicle network processor, and reported minimal impact on the boot flow [23]. The closest work is Oka and Jungklass, who measured ML-DSA, FN-DSA, and SLH-DSA verification for secure boot on the AURIX TC397XX and TC277TF, using the on-chip HSM as a minimal root of trust and delegating verification to the main cores in a staged boot [14]. Because signing and key generation run offline on a provisioning host, these works agree that verification latency and on-chip memory are the binding constraints for deployment [15].

The present work differs from these in three respects. It evaluates three ML-DSA security levels, both Falcon variants, both HAWK variants, and two classical as baselines. It runs entirely in software on a single core of the TC375TP, without the HSM and without hardware acceleration, and it adds a TriCore assembly port of the ML-DSA transform, an optimization that Oka and Jungklass list as future work. A signature-scheme evaluation of this breadth for secure boot on the AURIX TriCore family, in software and with an assembly-optimized transform, remains open; this thesis provides it.

1.5 Contributions

This thesis makes four contributions.

1. A bare-metal secure bootloader for the Infineon AURIX TC375TP. The bootloader implements a dual-slot firmware update policy, an 8 KiB signed metadata header for the public key and signature, and an Ethernet upload channel using the lwIP stack [24]. Chapter 3 describes the design.

2. The integration of three post-quantum signature schemes (ML-DSA, Falcon, and HAWK) and two classical baselines (Ed25519 and RSA-PSS), across their parameter variants, behind a single verification interface. The implementation also includes a host-side signing tool and trust-anchor provisioning.

3. A TriCore assembly implementation of the ML-DSA Number Theoretic Transform and its inverse. This adapts the ARM Cortex-M4 pqm4 strategy to TriCore address registers. The optimization reduces the cost of the transform in isolation by 94.1 %; its smaller effect on the complete sign and verify operations is quantified in Chapter 4.

4. A benchmark of all configurations on the target silicon. The evaluation reports cycle counts for key generation, signing, and verification, alongside rejection statistics, peak stack, code size, and key and signature sizes. The results indicate that post-quantum verification is feasible on this microcontroller in software, and that hashing the firmware dominates the verification latency.

1.6 Thesis Outline

This thesis is structured into the following chapters:

- **Chapter 2** presents the post-quantum background. It describes the quantum algorithms that threaten classical cryptography, the NIST standardization process and the lattice-based signature schemes evaluated in this work: ML-DSA, Falcon, and HAWK.
- **Chapter 3** details the target hardware and the secure bootloader implementation: the AURIX TC375TP platform and its memory layout, the system architecture, the signed header format, the boot decision procedure, the cryptographic verification path, the optimized NTT, and the host signing tool.
- **Chapter 4** presents the measurement methodology and the performance evaluation of all signature configurations on the target silicon, covering timing, memory footprint, and the impact of the NTT optimization.
- **Chapter 5** summarizes the findings, states the limitations of the prototype, and outlines future work.

Chapter 2

Post Quantum Cryptography

This chapter reviews the cryptographic background of the thesis. It first describes the quantum algorithms that threaten classical public-key cryptography, then the NIST standardization process and its security levels, and finally the three lattice-based signature schemes evaluated in this work: ML-DSA, Falcon, and HAWK.

Public-key cryptography authenticates data and establishes shared secrets between parties that share no prior secret. Its security rests on mathematical problems that are efficient to compute in one direction but infeasible to invert. Two such problems underlie the asymmetric primitives deployed today. The Integer Factorization Problem (IFP) is the basis of the RSA cryptosystem: recovering the prime factors p and q of a public modulus $n = pq$ is infeasible for a sufficiently large n. The Discrete Logarithm Problem (DLP) is the basis of the Digital Signature Algorithm (DSA): given a generator g and a value g^x in a finite cyclic group, recovering the exponent x is infeasible [5]. Elliptic Curve Cryptography (ECC) instantiates the discrete logarithm over the group of points of an elliptic curve, the Elliptic Curve Discrete Logarithm Problem (ECDLP), and reaches equivalent security with smaller keys [25]. These problems have no known efficient classical solution [5], which is the foundation of current asymmetric security.

The implementation of scalable quantum computation undermines the mathematical hardness assumptions foundational to classical public-key cryptography [6, 5].

Known quantum algorithms reduce the hardness of both IFP and DLP to polynomial-time computations. Table 2.1 relates each classical security strength to the corresponding RSA modulus and elliptic-curve key sizes. All cryptographic

systems relying on asymmetric primitives must transition to Post-Quantum Cryptography (PQC) to remain resistant against quantum cryptanalysis. To facilitate this transition, the National Institute of Standards and Technology (NIST) initiated a standardization process to evaluate and select quantum-resistant public-key algorithms, specifically targeting key encapsulation mechanisms (KEMs) and digital signature schemes [5].

Classical Security Strength (bits)	**IFP (RSA) Modulus k (bits)**	**ECC Order f (bits)**
112	2048	224–255
128	3072	256–383
192	7680	384–511
256	15360	512+

Table 2.1: Classical security strength and the corresponding RSA modulus and elliptic-curve key sizes, based on NIST SP 800-57 Part 1 Revision 5 [25].

2.1 Quantum Computational Threats to Cryptographic Architectures

Three quantum algorithms represent the primary threats: Shor's algorithm [6], Grover's algorithm [26], and the Brassard-Høyer-Tapp (BHT) algorithm [27].

2.1.1 Shor's Algorithm and Public-Key Cryptography

Shor's algorithm solves the Integer Factorization Problem and the Discrete Logarithm Problem in polynomial time [6], and the reduction extends to the elliptic-curve variant ECDLP.

Both the IFP and the ECDLP reduce to a period-finding problem. Shor's algorithm solves this period-finding problem using Quantum Phase Estimation (QPE) and the Quantum Fourier Transform (QFT). This polynomial-time reduction eliminates the classical security margins of RSA and ECC. For this reason, neither cryptosystem provides security against a Cryptographically Relevant Quantum Computer (CRQC) with sufficient number of qubits [5].

2.1.2 Grover's Algorithm and Symmetric Security

Grover's algorithm solves unstructured search problems in $\mathcal{O}(\sqrt{N})$ quantum iterations, where N denotes the search space size [26]. The algorithm applies iterative

amplitude amplification to identify the target state. The number of iterations required to achieve a near-certain success probability is approximately $\frac{\pi}{4}\sqrt{N}$ [28]. Applied to symmetric key search, this halves the effective bit strength: AES-128 key recovery is reduced from 2^{128} to 2^{64} Grover iterations.

The practical threat remains constrained by the non-parallelizability of the algorithm. The quadratic speedup is not multiplicative; splitting the search space across independent Grover instances does not reduce the total circuit depth below $\mathcal{O}(\sqrt{N})$. Moreover, each Grover iteration requires sequential execution of the full AES oracle as a quantum circuit, making the total circuit depth far exceed 2^{64} gate operations.

2.1.3 The Brassard-Høyer-Tapp Algorithm

The Brassard-Høyer-Tapp (BHT) algorithm applies Grover's algorithm to birthday-bound collision search on hash functions [27]. Given an n-bit hash function output, the search space is $N = 2^n$. The BHT algorithm identifies a collision in $\mathcal{O}(2^{n/3})$ quantum queries. The algorithm proceeds in two phases: store and search. In the first phase, $\mathcal{O}(2^{n/3})$ input-hash pairs are precomputed and stored in a quantum random access memory (qRAM). In the second phase, Grover's search iterates over the input domain to identify a new input whose hash value matches any stored pair. For SHA-256 ($n = 256$), this imposes a qRAM requirement of $\mathcal{O}(2^{85})$ coherent quantum states, which exceeds the capability of current quantum hardware.

Primitive & Security Property	**Classical Complexity**	**Quantum Complexity**
Symmetric Cipher Key Recovery	$O(2^n)$ (Exhaustive Search)	$O(2^{n/2})$ (Grover's)
Preimage / 2nd Preimage Hash	$O(2^n)$ (Brute-force)	$O(2^{n/2})$ (Grover's)
Collision Resistance Hash	$O(2^{n/2})$ (Birthday Paradox)	$O(2^{n/3})$ (BHT)

Table 2.2: Computational complexity of classical vs. quantum attacks on symmetric and hash primitives.

2.1.4 NIST PQC Standardization

In 2016, NIST initiated a public competition to evaluate and standardize post-quantum cryptographic algorithms [29]. The primary phase of this standardization process concluded in August 2024 with the publication of three finalized Federal Information Processing Standards (FIPS):

- **ML-KEM** (FIPS 203) [30], which is a key encapsulation mechanism based on the Kyber algorithm.

- **ML-DSA** (FIPS 204) [31], which is a lattice-based digital signature algorithm based on the CRYSTALS-Dilithium algorithm.
- **SLH-DSA** (FIPS 205) [18], which is a stateless hash-based digital signature algorithm based on the SPHINCS$^+$ algorithm.

Additionally, Falcon [32] was selected for standardization; the corresponding standard is designated as FN-DSA under FIPS Publication 206 [33], which is currently awaiting the release of its Initial Public Draft. To explore signature alternatives not based on structured lattices, NIST launched an additional call for digital signature schemes [34]. HAWK [35] advanced to the third round of this additional process [36].

2.1.5 NIST PQC Security Levels

Estimating the exact security strength of post-quantum cryptosystems is subject to two primary sources of uncertainty. First, the field of quantum algorithm research continues to develop; future algorithms may reduce the security margins of currently proposed schemes. Second, the performance characteristics of future quantum hardware, including physical and logical qubit counts and gate complexities, remain difficult to predict. For these reasons, NIST defines post-quantum security classification using five levels, benchmarked against the computational cost of attacking established symmetric cryptographic primitives, rather than specifying exact bit-strength equivalents.

The NIST evaluation criteria define these five security levels as follows [29]:

Level 1: "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit key (e.g. AES128)"

Level 2: "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for collision search on a 256-bit hash function (e.g. SHA256/ SHA3-256)"

Level 3: "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 192-bit key (e.g. AES192)"

Level 4: "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for collision search on a 384-bit hash function (e.g. SHA384/ SHA3-384)"

Level 5: "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g. AES 256)"

2.2 Selection of Evaluated Schemes

Digital signature scheme candidates were drawn from the submissions to the NIST PQC standardization competition and its additional call for signature schemes [29, 34].

- ML-DSA [31] was selected on the basis of its finalized standardization as FIPS 204, its moderate signature size, and its integer-only arithmetic, which reduces implementation complexity on embedded targets.
- Falcon [32] was selected for its compact signature size and fast verification.
- HAWK [35] was selected as an integer-only alternative to Falcon with comparable cryptographic material sizes, sourced from the NIST additional digital signature scheme competition [34].

Two classical schemes are included as baselines for comparison. Ed25519 represents the elliptic-curve signatures in current use and gives a best-case classical verification cost. RSA-PSS [37], at 2048 and 3072 bits, represents the scheme most widely fielded in deployed secure boot; it is evaluated at two public exponents ($e = 3$ and $e = 65537$) because the exponent determines the verification cost. Neither classical scheme resists a quantum adversary; they serve only to quantify the cost of the post-quantum migration.

2.3 ML-DSA

The Module-Lattice-Based Digital Signature Algorithm (ML-DSA) is a stateless digital signature scheme standardized by NIST in the Federal Information Processing Standards (FIPS) Publication 204 [31]. The algorithm was submitted as CRYSTALS-Dilithium [38] to the first round of the Post-Quantum Cryptography (PQC) standardization process initiated by NIST in 2016, and was selected for standardization in 2022 at the conclusion of the third round. ML-DSA is designed to be easier to implement securely than alternative lattice-based schemes, and uses uniform sampling instead of high-precision Gaussian sampling. This simplifies code and mitigates side-channel attacks during key generation and signing. However, this simplification increases the signature size.

ML-DSA instantiates its algebraic elements as polynomials within the quotient ring:

$$\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1) \tag{2.1}$$

The scheme defines a fixed prime modulus $q = 2^{23} - 2^{13} + 1 = 8{,}380{,}417$. The polynomial degree is set to $n = 256$ in all security levels defined by the specification documents.

ML-DSA is defined in three parameter sets, summarized in Table 2.3. The modulus q, the degree n, the root of unity $\zeta = 1753$, and the number of dropped low-order bits $d = 13$ are the same for all three. The matrix dimensions (k, l), the secret coefficient range η, the challenge weight τ, and the masking bound γ_1 grow with the security level, which also changes the mean number of signing attempts.

Table 2.3: ML-DSA parameter sets. The modulus $q = 8\,380\,417$, the degree $n = 256$, the root of unity $\zeta = 1753$, and the dropped-bit count $d = 13$ are identical across the three sets. `try` is the mean number of signing attempts.

Set	NIST level	$(\mathbf{k}, \mathbf{l})$	$\boldsymbol{\eta}$	$\boldsymbol{\tau}$	$\boldsymbol{\gamma_1}$	`try`
ML-DSA-44	2	(4, 4)	2	39	2^{17}	4.25
ML-DSA-65	3	(6, 5)	4	49	2^{19}	5.10
ML-DSA-87	5	(8, 7)	2	60	2^{19}	3.85

The security of ML-DSA reduces to two module-lattice hardness assumptions: the Module Learning With Errors (MLWE) problem and the Module Short Integer Solution (MSIS) problem [31].

The MLWE problem underlies the secrecy of the private key vectors. Given a public matrix $\mathbf{A} \in R_q^{k \times l}$ and a public vector $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \in R_q^k$, recovering the secret vector $\mathbf{s}$ is the problem of solving a set of noisy linear equations. Without

the error term $\mathbf{e}$, the relation $\mathbf{b} = \mathbf{As}$ would be an ordinary linear system over R_q and could be solved directly; the added noise makes the problem intractable for the best known techniques, including Gaussian elimination [31]. The error vector $\mathbf{e}$ is kept secret from the adversary; its polynomial coefficients satisfy $\|\mathbf{e}\|_\infty \leq \eta$, where $\eta \in \{2, 4\}$ is determined by the selected parameter set.

The MSIS problem secures the authenticity of the signatures. This problem states that finding a non-zero vector $\mathbf{v}$ such that $\mathbf{Av} = \mathbf{0}$ under a given bounded infinity norm $\|\mathbf{v}\|_\infty \leq \beta$ is computationally infeasible using currently known reduction techniques. The infinity norm $\|\mathbf{v}\|_\infty$ of a vector of polynomials represents the maximum absolute coefficient across all elements.[1]

2.3.1 Number Theoretic Transform

ML-DSA performs polynomial multiplication in the ring $\mathcal{R}_q$ repeatedly, in key generation, signing, and verification. Naive multiplication of two degree-n polynomials costs $\mathcal{O}(n^2)$ coefficient multiplications; for $n = 256$ this is 65,536 modular multiplications per product. The Number Theoretic Transform (NTT) reduces this cost.

The NTT is the finite-field analogue of the Discrete Fourier Transform. It is a ring isomorphism that maps a polynomial $a \in \mathcal{R}_q$ to its vector of evaluations $\hat{a} \in \mathbf{T}_q$ at fixed roots of unity. The isomorphism turns the convolution that underlies polynomial multiplication into a coefficient-wise product:

$$\widehat{a \cdot b} = \hat{a} \circ \hat{b}, \tag{2.2}$$

where $\circ$ is the coefficient-wise product. A product in $\mathcal{R}_q$ is therefore computed by transforming both operands, multiplying their evaluation vectors pointwise, and applying the inverse transform to the result. The forward and inverse transforms each cost $\mathcal{O}(n \log n)$ and the pointwise product costs $\mathcal{O}(n)$, so a single multiplication costs $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$. The advantage in practice: the public matrix $\hat{\mathbf{A}}$ and several intermediate vectors are kept directly in the NTT domain, so their transform is computed once and reused across many products.

The transform exists because of the chosen modulus. ML-DSA fixes $q = 8{,}380{,}417$, which satisfies $q \equiv 1 \pmod{512}$; this guarantees a primitive 512-th root of unity $\zeta = 1753$ in $\mathbb{Z}_q$. Because the ring modulus is $x^{256} + 1$, the transform evaluates a polynomial at the odd powers of ζ, which yields a negacyclic NTT. The evaluation points are $\zeta_i = \zeta^{2\,\mathrm{BitRev}_8(i)+1} \bmod q$ for $i = 0, \ldots, 255$, where $\mathrm{BitRev}_8(i)$

[1] For a single polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$, the norm is defined as $\|a\|_\infty = \max_i |a_i|$.

reverses the eight bits of i. The forward transform is

$$\mathrm{NTT}(a)_i = \sum_{j=0}^{255} a_j \, \zeta_i^j \pmod{q}. \tag{2.3}$$

FIPS 204 specifies an in-place iterative implementation: the Cooley-Tukey butterfly for the forward transform and the Gentleman-Sande butterfly for the inverse, with the powers of ζ held in a precomputed lookup table. This structure, and the register reuse it allows, is what the TriCore assembly port in Chapter 3 (Section 3.3.8) optimizes.

2.3.2 ML-DSA Signature Protocol

ML-DSA adapts the Schnorr signature paradigm under the Fiat-Shamir with aborts framework. A Schnorr signature is a Sigma protocol (Σ-protocol) bound to a message and made non-interactive by the Fiat-Shamir transformation. The same construction is used for the construction of Ed25519 Elliptic curve signature scheme, the classical baseline evaluated in Chapter 4.

Sigma Protocol

A Sigma protocol (Σ-protocol) is a three-move interactive proof of knowledge between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. Before starting the protocol, the prover generates a key pair. This pair consists of a private key w and a public key x linked by a mathematical relationship. The prover discloses x to the verifier but retains w as confidential. The prover aims to demonstrate knowledge of w to the verifier without revealing the key itself. This mechanism is known as a proof of knowledge.

The protocol consists of three sequential phases: commitment, challenge, and response.

1. **Commitment Phase:** The prover generates a random, uniformly distributed ephemeral secret r. The prover then computes a public commitment a from r and sends it to the verifier.

2. **Challenge Phase:** The verifier samples a random challenge c and transmits it to the prover. The verifier must choose c after receiving the commitment a. This strict ordering prevents an adversary from constructing valid proofs without knowledge of the secret key. Without this constraint, the prover could forge a valid response by pre-selecting c without knowing the private key.

3. **Response Phase:** The prover computes a response z. This value algebraically binds the challenge c, the private key w, and the ephemeral secret r. The prover sends z to the verifier.

The verifier then checks the algebraic consistency of the triple (a, c, z) against the public key x. The proof is accepted if and only if the verification relation holds; it is rejected otherwise.

In its interactive form, the Sigma protocol cannot generate digital signatures directly because the process does not sign an external message. Furthermore, both parties must remain online to exchange messages in real time. To prevent forgery, the prover must not learn the challenge before generating the commitment. However, a practical digital signature scheme requires non-interactive verification. Any third party must be able to verify the signature offline at any time.

Fiat-Shamir Heuristic

The Fiat-Shamir transformation converts an interactive proof of knowledge into a non-interactive signature scheme. Amos Fiat and Adi Shamir first described this method in 1986 [39]. They applied the transformation to a three-move interactive identification protocol based on the integer factorization problem. Ronald Cramer formally defined this three-move structure as a Sigma protocol in his 1996 PhD thesis [40].

The transformation replaces the interactive verifier with a cryptographic hash function. This function is modeled as a random oracle. It takes the public commitment a and the message m as inputs to produce the challenge:

$$c = H(a \parallel m) \tag{2.4}$$

To prevent forgery, this hash function must behave like a random oracle. Specifically, the prover must not be able to predict the challenge or manipulate the commitment to obtain a desired output. This model is known as the Random Oracle Model (ROM) [39].

The hash function must compute the challenge from both the commitment a and the message m. If the challenge depended only on the message, the scheme would be vulnerable to a forgery attack. An adversary possessing only the signer's public key could forge a valid signature for an arbitrary message without the private key.

To sign a message, the signer generates the commitment a and the response z. The resulting digital signature is the tuple (a, z). In their original paper, Fiat and Shamir represented the signature as (c, z). They chose this representation because the challenge c is smaller than the commitment a. During verification, the verifier can reconstruct the commitment a using c and z. This optimization reduces the overall signature size.

The Fiat-Shamir transformation eliminates the interactive verifier: signature verification requires only the public key, the message, and the signature tuple (a, z).

2.3.3 ML-DSA Operations

This section describes the three ML-DSA operations: key pair generation, signature generation, and signature verification.

Key pair generation

During key pair generation, a 32-byte seed ξ is randomly drawn and expanded using SHAKE256[2] to generate a 32-byte public seed ρ, a 64-byte noise seed ρ', and a 32-byte private key seed K. The public seed ρ is used to generate the public matrix $\mathbf{A} \in \mathcal{R}_q^{k \times l}$. To optimize performance, the algorithm expands ρ to sample the coefficients of $\mathbf{A}$ directly in the NTT domain as $\hat{\mathbf{A}}$ using the function `polyvec_matrix_expand`. The noise seed ρ' is used to generate the secret vectors $\mathbf{s}_1 \in \mathcal{R}_q^l$ and $\mathbf{s}_2 \in \mathcal{R}_q^k$ through the functions `polyvecl_uniform_eta` and `polyveck_uniform_eta`, respectively. The coefficients of these vectors are bounded by the parameter η, such that they lie in the range $[-\eta, \eta]$. The private key seed K is used during the signature phase. Then, the public vector $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 \in \mathcal{R}_q^k$ is computed and then decomposed into (t1,t0) using the function Power2Round as $\mathbf{t} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$. The tuple $pk = (\rho, \mathbf{t}_1)$ is the public key. The secret key is the tuple $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, where $tr = H(pk,64)$.

In the code implementation, $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ (line 5) is performed in the NTT domain. For this reason $\mathbf{s}_1$ is transformed to NTT domain allowing the pointwise Montgomery multiplication using the function `polyvec_matrix_pointwise_montgomery`. During this accumulation, the implementation uses lazy reduction since the modulus $q = 8380417$ is a 23 bits prime and the computation are done in 32 bits registers, coefficient additions can accumulate without requiring modular reduction at each step. The function `polyveck_reduce` is invoked to reduce the coefficients after the matrix-vector product due to the fact the inverse NTT (`polyveck_invntt_tomont`) requires its input to strictly be bounded modulo q. The sum with vector $\mathbf{s}_2$ is performed using the function `polyvec_add` and then the coefficients of the result are normalized in the range $[0, q-1]$ using the function `polyveck_caddq`.

[2]FIPS 204 derives these seeds as $H(\xi \,\|\, \text{IntegerToBytes}(k,1) \,\|\, \text{IntegerToBytes}(\ell,1), 128)$, where H is SHAKE256 and the two appended bytes provide domain separation between parameter sets; the listing omits them for readability. The implementation follows the standard.

Algorithm 1 ML-DSA Key Pair Generation

Output: Public key $pk = (\rho, \mathbf{t}_1)$, Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

1: $\xi \leftarrow \mathbb{B}^{32}$ ▷ Sample random seed
2: $(\rho, \rho', K) \leftarrow \text{SHAKE256}(\xi \,\|\, k \,\|\, l, 128)$ ▷ Seed expansion, k and l as single bytes
3: $\hat{\mathbf{A}} \in \mathbf{T}_q^{k \times l} \leftarrow \text{ExpandA}(\rho)$ ▷ Generate matrix in NTT domain
4: $(\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{R}_q^l \times \mathcal{R}_q^k \leftarrow \text{ExpandS}(\rho')$ ▷ Sample short error vectors
5: $\mathbf{t} \leftarrow \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ ▷ Compute public target vector
6: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}(\mathbf{t})$ ▷ Compress public vector
7: $tr \leftarrow \text{SHAKE256}(pk, 64)$ ▷ Hash public key for binding
8: **return** $pk = (\rho, \mathbf{t}_1)$, $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

Signature generation

The signature generation process operates on a message representative μ derived from the public key hash and the message. Following the Schnorr signature paradigm, the protocol comprises three phases: commitment, challenge, and response.

As initialization, the matrix $\mathbf{A}$ is expanded using the seed ρ, vectors $\mathbf{s}_1$ and $\mathbf{s}_2$ recovered from the secre material bundle and both transformed in NTT domai, right before the main loop.

During the commitment phase, the signer pseudo-randomly generates a secret masking vector $\mathbf{y} \in \mathcal{R}_q^l$ with coefficients in the interval $[-\gamma_1+1, \gamma_1]$ then transformed in NTT domain. To compute the intermediate vector $\mathbf{w} = \mathbf{A}\mathbf{y}$, $\mathbf{y}$ is transformed in NTT domain, multiplied by $\hat{\mathbf{A}}$, and obtained via inverse NTT. The public commitment vector $\mathbf{w}_1 = \text{HighBits}(\mathbf{w})$ is the vector of high-order bits of $\mathbf{w}$. This extraction rounds $\mathbf{w}$ according to the power-of-two decomposition parameter $2\gamma_2$.

The commitment hash $\tilde{c} \in \mathbb{B}^{\lambda/4}$ is computed as $H(\mu \,\|\, \text{w1Encode}(\mathbf{w}_1), \lambda/4)$. From $\tilde{c}$, SampleInBall samples a challenge polynomial $c \in \mathcal{R}_q$ with exactly τ non-zero ternary coefficients in $\{-1, 0, 1\}$. This step corresponds to `poly_challenge`, which consumes the digest $\tilde{c}$ to generate the sparse polynomial c.

In the response phase, the signer computes the candidate response vector $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ and the low-order residual $\mathbf{r}_0 = \text{LowBits}(\mathbf{w} - c\mathbf{s}_2,\, 2\gamma_2)$. If $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$, the rejection sampling loop restarts with a fresh masking vector $\mathbf{y}$.

If the bounds hold, the signer computes a hint vector $\mathbf{h}$, which enables the reconstruction of $\mathbf{w}_1$ by the verifier. The hint vector is defined as:

$$\mathbf{h} = \text{MakeHint}(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$$

where $\mathbf{t}_0$ encodes the d low-order bits of $\mathbf{t}$. The signature candidate is additionally rejected if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or the Hamming weight of $\mathbf{h}$ exceeds ω.

After obtaining a valid signature a final output $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$ is returned. In the C implementation, computations and the validation steps are done in a fail-early manner in four different point in time, sequentially. First, $\mathbf{z}$ is computed by multiplying c and $\mathbf{s}_1$ (both already in the NTT domain), computed the inverse NTT and vector $\mathbf{y}$ is added to the result; the check of the norm of $\mathbf{z}$ done right after and the loop restarted if out of bound. Second, c and $\mathbf{s}_2$ (both in NTT) are multiplied and converted back via the inverse NTT, then subtracted directly from $\mathbf{w}_0$ to compute $\mathbf{r}_0$ and its norm checked via `polyveck_chknorm`, loop is restarted if the norm is out of bound. Third, the product $c\mathbf{t}_0$ is computed in the same manner, its norm checked against γ_2. Finally the hint vector $\mathbf{h}$ is generated via `polyveck_make_hint`, and the attempt is rejected if its number of non-zero bits exceeds ω.

Algorithm 2 ML-DSA Signature Generation

Input: Secret key $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, message M
Output: Signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

$\hat{\mathbf{A}} \in \mathbf{T}_q^{k \times l} \leftarrow \text{ExpandA}(\rho)$ ▷ Regenerate matrix in NTT domain
$\mu \leftarrow \text{SHAKE256}(tr \parallel \texttt{0x00} \parallel \texttt{0x00} \parallel M, 64)$ ▷ Message representative
$\rho'' \leftarrow \text{SHAKE256}(K \parallel rnd \parallel \mu, 64)$ ▷ Derive masking seed
$\kappa \leftarrow 0$ ▷ Initialize attempt counter
$(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do** ▷ Rejection sampling loop
    $\mathbf{y} \in \mathcal{R}_q^l \leftarrow \text{ExpandMask}(\rho'', \kappa)$ ▷ Sample masking vector
    $\mathbf{w} \leftarrow \mathbf{A}\mathbf{y}$ ▷ Compute commitment
    $\mathbf{w}_1 \leftarrow \text{HighBits}(\mathbf{w})$ ▷ Extract high-order bits
    $\tilde{c} \leftarrow \text{SHAKE256}(\mu \parallel \text{w1Encode}(\mathbf{w}_1), \lambda/4)$ ▷ Commitment hash
    $c \in \mathcal{R}_q \leftarrow \text{SampleInBall}(\tilde{c})$ ▷ Derive challenge polynomial
    $\mathbf{z} \leftarrow \mathbf{y} + c\mathbf{s}_1$ ▷ Compute response
    $\mathbf{r}_0 \leftarrow \text{LowBits}(\mathbf{w} - c\mathbf{s}_2)$ ▷ Low-order residual
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
        $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ Reject and restart
    else
        $\mathbf{h} \leftarrow \text{MakeHint}(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$ ▷ Compute hint vector
        if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** $\text{HammingWeight}(\mathbf{h}) > \omega$ **then**
            $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ Reject and restart
        end if
    end if
    $\kappa \leftarrow \kappa + l$ ▷ Advance counter
end while
return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

Signature Verification

The verification phase reconstructs the commitment vector $\mathbf{w}_1$ using only the public key $pk = (\rho, \mathbf{t}_1)$ and the signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$. The challenge polynomial c is recomputed from $\tilde{c}$ via SampleInBall to match the value computed during signing. The verification algorithm then computes an approximation of the commitment vector,

$$\mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d,$$

and applies the hint vector $\mathbf{h}$ to recover the rounded commitment,

$$\mathbf{w}'_1 = \text{UseHint}(\mathbf{h},\ \mathbf{w}'_{\text{Approx}}).$$

Finally, the algorithm computes the candidate commitment hash:

$$\tilde{c}' = H(\mu \parallel \text{w1Encode}(\mathbf{w}'_1),\ \lambda/4).$$

The signature is valid if and only if both of the following conditions are satisfied:

$$\tilde{c} = \tilde{c}' \qquad \text{and} \qquad \|\mathbf{z}\|_\infty < \gamma_1 - \beta.$$

As in the previous operations, the actual source code computes $\mathbf{w}'_{\text{Approx}}$ by transforming the elements into the NTT domain, performing the operations, and transforming the result back via the inverse NTT. The left-shift of the coefficients of $\mathbf{t}_1$ is applyed before its NTT transformation. Finally, `polyveck_use_hint` decomposes each coefficient of $\mathbf{w}'_{\text{approx}}$ and uses the bit flags in $\mathbf{h}$ to correct the high-order portion of the coeffitient by ± 1 whenever the error crosses a rounding boundary.

The verification is based on the reconstruction of the high-order bits $\mathbf{w}_1$ from an approximation of the commitment vector $\mathbf{w}$. During signature generation, the commitment vector was defined as $\mathbf{w} = \mathbf{A}\mathbf{y}$. In the response phase, the signer computed the response vector $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$, hence $\mathbf{y} = \mathbf{z} - c\mathbf{s}_1$. Furthermore, the public key is $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, so $\mathbf{A}\mathbf{s}_1 = \mathbf{t} - \mathbf{s}_2$. By replacing the terms, the commitment vector $\mathbf{w}$ can be expressed in terms of the public key and response vector:

$$\begin{aligned} \mathbf{w} &= \mathbf{A}(\mathbf{z} - c\mathbf{s}_1) \\ &= \mathbf{A}\mathbf{z} - c(\mathbf{A}\mathbf{s}_1) \\ &= \mathbf{A}\mathbf{z} - c(\mathbf{t} - \mathbf{s}_2) \\ &= \mathbf{A}\mathbf{z} - c\mathbf{t} + c\mathbf{s}_2 \end{aligned} \tag{2.5}$$

The verifier does not possess the full public vector $\mathbf{t}$, but only the high-order bits $\mathbf{t}_1$ extracted via the power-of-two decomposition $\mathbf{t} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$. Then, the verifier computes the approximation vector $\mathbf{w}'_{\text{approx}}$ as:

$$\mathbf{w}'_{\text{approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d \tag{2.6}$$

The difference between the exact commitment vector and this approximation yields:

$$\begin{aligned} \mathbf{w} - \mathbf{w}'_{\text{approx}} &= (\mathbf{Az} - c\mathbf{t} + c\mathbf{s}_2) - (\mathbf{Az} - c\mathbf{t}_1 \cdot 2^d) \\ &= c\mathbf{s}_2 - c(\mathbf{t} - \mathbf{t}_1 \cdot 2^d) \\ &= c\mathbf{s}_2 - c\mathbf{t}_0 \end{aligned} \tag{2.7}$$

By construction, c is a vector with small coefficients $\{-1, 0, +1\}$ containing exactly τ non-zero coefficients and s_2 is a vector with small coefficients in the range $[-\eta, +\eta]$, where $\eta = 2$ or 4, depending on the parameter set.

The low-order public key vector $\mathbf{t}_0$ contains coefficients bounded within the range $[-2^{d-1} + 1, 2^{d-1}]$.

Because the difference $\mathbf{w} - \mathbf{w}'_{\text{approx}}$ is small enough, the verifier can use the hint vector $\mathbf{h}$ to correct the approximation and reconstruct the high-order bits $\mathbf{w}_1$ originally computed during the commitment phase.

Algorithm 3 ML-DSA Signature Verification

Input: Public key $pk = (\rho, \mathbf{t}_1)$, message M, signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
Output: Boolean

1: $\hat{\mathbf{A}} \in \mathbf{T}_q^{k \times l} \leftarrow \text{ExpandA}(\rho)$ ▷ Regenerate matrix in NTT domain
2: $tr \leftarrow \text{SHAKE256}(pk, 64)$ ▷ Recompute public key hash
3: $\mu \leftarrow \text{SHAKE256}(tr \parallel \texttt{0x00} \parallel \texttt{0x00} \parallel M, 64)$ ▷ Message representative
4: $c \in \mathcal{R}_q \leftarrow \text{SampleInBall}(\tilde{c})$ ▷ Recover challenge polynomial
5: $\mathbf{w}'_{\text{Approx}} \leftarrow \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d$ ▷ Approximate the commitment
6: $\mathbf{w}'_1 \leftarrow \text{UseHint}(\mathbf{h}, \mathbf{w}'_{\text{Approx}})$ ▷ Correct it with the hint
7: $\tilde{c}' \leftarrow \text{SHAKE256}(\mu \parallel \text{w1Encode}(\mathbf{w}'_1), \lambda/4)$ ▷ Recompute commitment hash
8: **return** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ **and** $\tilde{c} = \tilde{c}'$ ▷ Accept only if both hold

Pre-Hash Variant (HashML-DSA)

FIPS 204 also defines a pre-hash variant called HashML-DSA. In pure ML-DSA the signer hashes the full message with SHAKE-256 inside the signing operation. When the message is large, or when the platform provides hardware support for a hash function other than SHAKE-256, this step can dominate the cost. HashML-DSA addresses this by signing a digest of the message, computed with a chosen hash function or extendable-output function and combined with domain-separation information, rather than the message itself [31]. Key generation is identical to pure ML-DSA; only the signing and verification algorithms differ. The standard states that the pure variant is preferred by default. The relevance of the pre-hash variant to firmware verification, where the signed message is the full application binary, is discussed in Chapter 5.

2.4 Falcon

Falcon [32] is a lattice-based digital signature algorithm whose security is based on the hardness of three problems over NTRU lattices: the Short Integer Solution (SIS) problem, the Shortest Vector Problem (SVP), and the Closest Vector Problem (CVP). Section 2.4.3 defines these problems and maps each one to the part of the scheme it protects. The acronym stands for Fast Fourier lattice-based compact signatures over NTRU. While Falcon was selected during the NIST Post-Quantum Cryptography (PQC) standardization process, the final standard is designated as FN-DSA under FIPS Publication 206.

Given the size of the cryptographic material in other proposed post quantum schemes, Falcon aims to minimize the size of the public key and the signature.

It is based on three components: the Gentry-Peikert-Vaikuntanathan (GPV) framework, which provides the hash-and-sign structure for lattice-based signatures; NTRU lattices, which instantiate the public and secret bases concretely; and Fast Fourier sampling, which performs the trapdoor sampling during signing.

2.4.1 GPV framework

The GPV framework describes how to build hash-and-sign lattice-based signature schemes that are secure in the classical and quantum random oracle models. In the general GPV framework, the public key is a full-rank matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ that generates a lattice Λ. The secret key is a matrix $\mathbf{B} \in \mathbb{Z}_q^{m \times m}$ that generates an orthogonal lattice to Λ, satisfying $\mathbf{B}\mathbf{A}^T = \mathbf{0}$.

A valid signature for a message mapped to a vector $\mathbf{h} \in \mathbb{Z}_q^n$ requires a short vector $\mathbf{s} \in \mathbb{Z}_q^m$ such that $\mathbf{s}\mathbf{A}^T = \mathbf{h}$. To compute this short vector $\mathbf{s}$, the signing algorithm first calculates a generic vector $\mathbf{c}$ satisfying $\mathbf{c}\mathbf{A}^T = \mathbf{h}$.

The algorithm then uses the matrix $\mathbf{B}$ to determine a vector $\mathbf{v}$ in the orthogonal lattice that is close to $\mathbf{c}$; this is an instance of the Closest Vector Problem (CVP), discussed in Section 2.4.3. The final signature vector $\mathbf{s}$ is the difference $\mathbf{s} = \mathbf{c} - \mathbf{v}$, assuming the distance between $\mathbf{c}$ and $\mathbf{v}$ satisfies the required bound.

2.4.2 NTRU lattices

To define the structure of matrices $\mathbf{A}$ and $\mathbf{B}$, Falcon utilizes the NTRU framework. In the NTRU framework, the secret material consists of four polynomials f, F, g, and G in $\mathbb{Z}[x]/(\phi)$, where $\phi = x^n + 1$. These polynomials must satisfy the NTRU equation $fG - gF = q \pmod{\phi}$, where $q = 12289$ is prime and f is invertible modulo q. The public material is the polynomial h, defined as $h = gf^{-1} \pmod{q}$. In the classic instantiation of NTRU lattices, the public matrix $\mathbf{A}$ is structured as $\begin{bmatrix} 1 & h \\ 0 & q \end{bmatrix}$ and the secret matrix $\mathbf{B}$ is structured as $\begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$.

The signature is the vector $\mathbf{s} = (s_1, s_2)$, which the verification algorithm checks against the required shortness bound. To compute such vector a technique by Ducas-Prest for the Fast Fourier Sampling Ducas-Prest is used in which via a data structure called Falcon tree and a target c obtaining the short vector s is fast and no information about the secret matrix is leaked.

One of the consequences of using a fast Fourier sampling technique in Falcon is the requirement of using floating-point arithmetic, opening the question on how many bits of precision are necessary to maintain the scheme's security guarantees. An evaluation based on a Renyi divergence argument demonstrates that 53 bits of precision are strictly sufficient. This level of precision, which corresponds to the mantissa of the standard IEEE 754 binary64 (double-precision) format, guarantees that the trapdoor sampler does not leak information about the secret basis for NIST parameters, specifically for a security level $\lambda \leq 256$ and up to $q_s \leq 2^{64}$ signature queries. For constrained devices that lack a native hardware support for floating point arithmetic, the reference implementation uses 64 bits integer operations to emulate the binary64 FPU computations.

2.4.3 Underlying hard problems

The security of Falcon is based on three problems over the NTRU lattice Λ. The Shortest Vector Problem (SVP) asks, given a basis of Λ, for a non-zero lattice vector of minimal Euclidean norm. The Closest Vector Problem (CVP) asks, given a basis and a target point $\mathbf{t}$ in the span of Λ, for the lattice vector closest to $\mathbf{t}$. The Short Integer Solution (SIS) problem asks, given the public matrix $\mathbf{A}$ and a bound β, for a short non-zero vector $\mathbf{z}$ with $\mathbf{z}\mathbf{A}^T = \mathbf{0} \pmod{q}$ and $\|\mathbf{z}\| \leq \beta$.

Each of these problems relates to a different part of the scheme. Recovering the secret basis $(g, f), (G, F)$ from the public key h is an SVP instance on Λ, since this basis consists of the short vectors that generate the lattice; its hardness is the NTRU assumption [32]. Signing is a CVP instance: the signer maps the message to a target $\mathbf{c}_0$ and computes a lattice vector close to it. The short secret basis $\mathbf{B}$ is the trapdoor that makes this CVP tractable; without it the same computation is infeasible. Forging a signature without the trapdoor is producing a short $\mathbf{s}$ with $\mathbf{s}\mathbf{A}^T = H(m)$, an SIS instance. The GPV framework is proven secure in the random oracle model under the hardness of SIS, adapted to the NTRU setting [32].

2.4.4 Falcon Operations

Falcon comes in two variant named Falcon-512 and Falcon-1024, respectively for NIST security requirement level 1 and 5. The principal elements in Falcon are polynomials of degree less than n in the ring $\mathbb{Z}_q[x]/(\phi)$, where $\phi = x^n + 1$. The

size of the degree is set to $n = 512$ for Falcon-512 and $n = 1024$ for Falcon-1024. The integer $q = 12289$ is the same for both variants. Verification multiplies polynomials modulo q through a Number Theoretic Transform that uses the generator $g = 7$, a primitive 2048-th root of unity modulo q. The forward zeta values are $\text{GMb}[x] = R\left(g^{\text{rev}(x)}\right) \bmod q$, where rev is the 10-bit bit-reversal, and the inverse transform uses $g^{-1} = 8778$. The factors are stored in a precomputed lookup table, as in ML-DSA.

Key pair generation

During key pair generation, the algorithm generates two polynomials f and g by sampling from a discrete Gaussian distribution that yield small coefficients. The algorithm then computes two additional polynomials F and G that satisfy the NTRU equation $fG - gF = q \pmod{\phi}$.

The secret key contains the matrix $\mathbf{B} = \left[\begin{smallmatrix} g & -f \\ G & -F \end{smallmatrix}\right]$, and the algorithm stores its Fast Fourier Transform (FFT) representation as sk. In this phase, the Falcon tree for the matrix $\mathbf{B}$ can be computed and stored in the secret key bundle to avoid the recomputation during the signature generation phase.

The public key is the polynomial h, computed as $h = gf^{-1} \pmod q$. These four polynomials instantiate the public matrix $\mathbf{A}$ and the secret matrix $\mathbf{B}$ in the form given in Section 2.4.2. The secret matrix $\mathbf{B}$ is a basis of short vectors for the NTRU lattice (Section 2.4.3). The public matrix $\mathbf{A}$ describes the same lattice as the set of integer vectors $\mathbf{x}$ satisfying $\mathbf{x}\mathbf{A}^T \equiv \mathbf{0} \pmod q$.

Algorithm 4 Falcon Key Pair Generation

Output: Secret key $sk = (\hat{\mathbf{B}}, \mathsf{T})$, public key $pk = h$
repeat
  $f, g \leftarrow D_{\mathbb{Z}, \sigma_{f,g}}$ ▷ Sample short polynomials
  $\gamma \leftarrow \|\mathbf{B}\|_{\text{GS}}$ ▷ Gram-Schmidt norm of the basis
  $(F, G) \leftarrow \text{NTRUSolve}(f, g)$ ▷ Solve $fG - gF = q$
until f is invertible mod q, $\gamma \leq 1.17\sqrt{q}$, and $(F, G) \neq \perp$
$\mathbf{B} \leftarrow \left[\begin{smallmatrix} g & -f \\ G & -F \end{smallmatrix}\right]$ ▷ Assemble the secret basis
$\hat{\mathbf{B}} \leftarrow \text{FFT}(\mathbf{B})$ ▷ Store the basis in FFT representation
$\mathsf{T} \leftarrow \text{ffLDL}^\star(\hat{\mathbf{B}}\hat{\mathbf{B}}^\star)$ ▷ Build the Falcon tree
for each leaf of T: leaf $\leftarrow \sigma/\sqrt{\text{leaf}}$ ▷ Normalize the tree leaves
$h \leftarrow gf^{-1} \bmod q$ ▷ Derive the public polynomial
return $sk = (\hat{\mathbf{B}}, \mathsf{T})$, $pk = h$

In PQClean's implementation of Falcon, key generation uses a fail-early approach because solving the NTRU equation to compute F and G is expensive. Three

checks are done before executing the `solve_NTRU` function.

Once the small polynomials f and g are sampled via `poly_small_mkgauss`, the implementation first checks the squared norm of $(g, -f)$ to discard oversized polynomials.

Then, the algorithm computes the Gram-Schmidt orthogonal vector to (f, g), $\tilde{\mathbf{b}}_2$, such that the area spanned by the two vectors is q, and checks if it is short enough. Because this orthogonal vector depends only on f, g, and q, we can evaluate its norm via FFT and discard unsuitable f, g pairs. A short orthogonal vector is a necessary condition for the existence of sufficiently small polynomials F and G satisfying the NTRU equation, ensuring that the resulting lattice basis is suitable for Falcon's signature sampler.

Next, the public component h is computed; this is also considered a check since the computation requires f to be invertible modulo q.

Finally, the resolution of the NTRU equation is attempted and F and G are produced; if it fails, the loop restarts. The standard PQClean key generation of Falcon does not compute and store the Falcon tree associated with the key pair; instead, it will be recomputed in the signature phase

Signature generation

The signature generation process begins by sampling a 40-byte nonce r from a uniform distribution and concatenating it with the message m. This randomization ensures message uniqueness and prevents the algorithm from signing identical messages multiple times; in the GPV framework, collecting multiple signatures for the same message can lead to information leaking about the secret basis matrix $\mathbf{B}$.

The algorithm hashes the concatenation $r \parallel m$ into a polynomial $c \in \mathbb{Z}_q[x]/(x^n + 1)$, constructing the target vector $\mathbf{c} = (c, 0)$. This operation implements the hash function within the hash-and-sign paradigm. Next, the algorithm computes the target vector in the Fast Fourier Transform (FFT) domain as:

$$\mathbf{t} = (\mathrm{FFT}(c), \mathrm{FFT}(0))\hat{\mathbf{B}}^{-1},$$

where $\mathbf{t}$ represents the target point $\mathbf{c}$ in lattice coordinates.

The signing algorithm then uses the Falcon tree T and the initial position $\mathbf{t}$ to sample a vector $\mathbf{z}$ via Fast Fourier sampling. The vector $\mathbf{z}$ constitutes an FFT representation of a lattice coordinate vector residing near the target point $\mathbf{t}$. The algorithm computes the difference vector $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$ and evaluates its norm against the required bound β^2. If the norm exceeds the bound, the algorithm discards the vector and samples a new $\mathbf{z}$. Upon successful bound validation, the algorithm computes the inverse FFT of $\mathbf{s}$ to recover the polynomial components s_1 and s_2. The final signature consists of the nonce r and the compressed polynomial s_2.

Algorithm 5 Falcon Signature Generation

Input: Message m, secret key $sk = (\hat{\mathbf{B}}, \mathsf{T})$, acceptance bound $\lfloor \beta^2 \rfloor$
Output: Signature $sig = (r, \mathbf{s})$

$r \leftarrow \{0,1\}^{320}$ ▷ Sample a 40 byte nonce
$c \leftarrow \text{HashToPoint}(r \,\|\, m)$ ▷ Hash the message to a point
$\mathbf{t} \leftarrow (\text{FFT}(c), \text{FFT}(0))\, \hat{\mathbf{B}}^{-1}$ ▷ Express the target in lattice coordinates
repeat
  repeat
    $\mathbf{z} \leftarrow \text{ffSampling}(\mathbf{t}, \mathsf{T})$ ▷ Sample random $\mathbf{z}$ using FFT
    $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$ ▷ Compute difference in std coordinates
  until $\|\mathbf{s}\|^2 \leq \lfloor \beta^2 \rfloor$ ▷ Reject if too long
  $(s_1, s_2) \leftarrow \text{invFFT}(\mathbf{s})$ ▷ Back to coefficient representation
  $\mathbf{s} \leftarrow \text{Compress}(s_2)$ ▷ Variable-length encoding
until $\mathbf{s} \neq \perp$ ▷ Reject if it does not fit the fixed size
return $sig = (r, \mathbf{s})$

Due to the nature of Falcon and the use of the Fast Fourier Transform, the first step converts the small integer polynomials of the secret basis into 64 bits floating point numbers using `smallints_to_fpr`. This is part of the floating point emulation mechanism. Then, the FFT is applied to the polynomials of the secret basis and the implementation computes the Gram matrix $G = \mathbf{B}\mathbf{B}^*$.

The sampling step is performed by `ffSampling_fft_dyntree`, which samples the lattice vector $\mathbf{z}$ using the target $\mathbf{t}$ and dynamically reconstructs the Falcon tree on the fly from the Gram matrix.

Signature verification

Verification reconstructs the target polynomial c by hashing $r \,\|\, m$. The polynomial s_2 is decompressed from the signature; the verifier rejects the signature if the decoding fails. The polynomial s_1 is not sent; it is reconstructed from c, s_2, and h via $s_1 = c - s_2 h \pmod{q}$. The relation $s_1 + s_2 h \equiv c \pmod{q}$ holds by construction of s_1. The signature is accepted if and only if $\|\mathbf{s}\|^2 \leq \beta^2$.

Algorithm 6 Falcon Signature Verification

Input: Message m, signature $sig = (r, \mathbf{s})$, public key $pk = h$, acceptance bound $\lfloor \beta^2 \rfloor$
Output: Boolean

$c \leftarrow \text{HashToPoint}(r \,\|\, m)$ ▷ Recompute the target point
$s_2 \leftarrow \text{Decompress}(\mathbf{s})$ ▷ Recover the signature polynomial
if $s_2 = \perp$ **then**
    return reject ▷ Invalid encoding
end if
$s_1 \leftarrow c - s_2 h \bmod q$ ▷ Reconstruct the missing component in $[-q/2, q/2]$
return $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ ▷ Accept only if the vector is short

For an honestly generated signature, the reconstruction recovers the signer's s_1; this follows directly from the lattice properties. Since $\mathbf{s} = \mathbf{t} - \mathbf{z}$ where $\mathbf{z} \in \Lambda(\mathbf{B})$, and given that the public matrix $\mathbf{A}$ satisfies $\mathbf{B}\mathbf{A}^T = \mathbf{0} \pmod q$, hence:

The source code follows the algorithm in a straightforward way. Since the polynomials involved in the signature verification have integer coefficients, the implementation uses the NTT for polynomial arithmetic over the ring $\mathbb{Z}_q[x]/(x^n+1)$, using $g = 7$ as the primitive 2048-th root of unity modulo 12289. In this manner this root can be used for both Falcon-512 and Falcon-1024. First, the public key polynomial h is deserialized from the public key and converted into the NTT domain. In the same manner, s_2 is decoded from the signature buffer, its coefficients are reduced into the interval $[0, q-1]$, and it is transformed via NTT. The product $s_2 h \pmod q$ is computed using Montgomery multiplication (`mq_poly_montymul_ntt`) and transformed back to the standard coefficient domain via `iNTT`. Instead of computing $s_1 = c - s_2 h$, the implementation subtracts the hashed message c obtaining $-s_1 = s_2 h - c \pmod q$. This avoids an unnecessary sign negation, because squaring the coefficients to compute the norm eliminates the minus sign ($\|(-s_1, s_2)\|^2 = \|(s_1, s_2)\|^2$). Finally, the coefficients of $-s_1$ are centered in the range $[-q/2, q/2]$ and the aggregate squared norm is checked against the acceptance bound $\lfloor \beta^2 \rfloor$.

$$s_1 + s_2 h = \begin{bmatrix} s_1 & s_2 \end{bmatrix} \begin{bmatrix} 1 \\ h \end{bmatrix} \pmod q$$

The source code follows the algorithm in a straightforward way. Since the

Using the full 2×2 public matrix $\mathbf{A} = \begin{bmatrix} 1 & h \\ 0 & q \end{bmatrix}$ and its transpose $\mathbf{A}^T = \begin{bmatrix} 1 & 0 \\ h & q \end{bmatrix}$,

we can state the vector product as $\mathbf{s}\mathbf{A}^T \pmod q$:

$$\begin{aligned}
\mathbf{s}\mathbf{A}^T &= (\mathbf{t}-\mathbf{z})\mathbf{B}\mathbf{A}^T \pmod q \\
&= \mathbf{t}\mathbf{B}\mathbf{A}^T - \mathbf{z}\mathbf{B}\mathbf{A}^T \pmod q \\
&= \mathbf{t}\mathbf{B}\mathbf{A}^T - \mathbf{0} \pmod q \qquad \mathbf{z}\mathbf{B}\mathbf{A}^T \text{ is zero: } \mathbf{B} \text{ and } \mathbf{A}^T \text{ are orthogonal and } \mathbf{z} \text{ is an integer polynomial} \\
&= \begin{bmatrix} c & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ h & q \end{bmatrix} \pmod q \\
&= \begin{bmatrix} c & 0 \end{bmatrix} \pmod q
\end{aligned}$$

Expanding the product $\mathbf{s}\mathbf{A}^T$ with $\mathbf{A}^T$ yields:

$$\begin{bmatrix} s_1 & s_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ h & q \end{bmatrix} = \begin{bmatrix} s_1 + s_2 h, & s_2 q \end{bmatrix} \pmod q$$

Equating the first components of both expansions leads to the verification relation:

$$s_1 + s_2 h \equiv c \pmod q$$

Figure 2.1 illustrates a simplified two-dimensional geometric representation of the signing process in Falcon. This low-dimensional visualization serves as an intuitive toy model and does not represent the high-dimensional polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$ with $n \in \{512, 1024\}$ utilized in the actual scheme. Only the signer holds the private basis $B = \{\mathbf{b}_1, \mathbf{b}_2\}$ (shown as short, nearly orthogonal blue vectors), whereas the public basis $A = \{\mathbf{a}_1, \mathbf{a}_2\}$ (shown as long, skewed orange vectors) is public. When signing, the signer maps the hashed message to a target vector $\mathbf{c} = (c, 0)$ and uses the private basis B to sample a nearby lattice point $\mathbf{z}$ from a discrete Gaussian distribution centered at $\mathbf{c}$ with standard deviation σ. The signature is the short difference vector $\mathbf{s} = \mathbf{c} - \mathbf{z}$. Using only the public basis A, the verifier checks that $\mathbf{z} = \mathbf{c} - \mathbf{s}$ lies on the lattice and that the length of $\mathbf{s}$ falls within the boundary, without revealing the private basis.

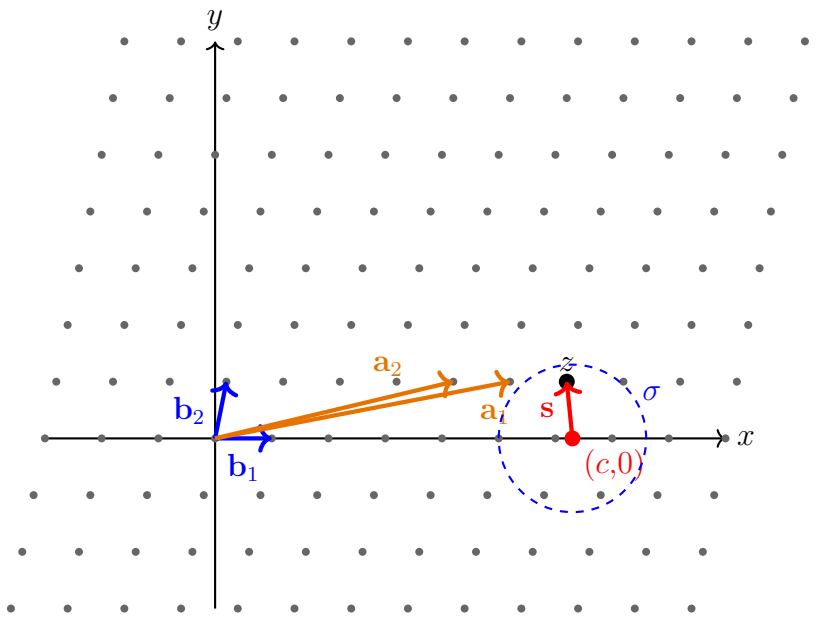


Figure 2.1: Two-dimensional toy representation of the Falcon signature scheme geometry (not to scale).

2.5 HAWK

HAWK is a lattice-based hash-and-sign signature scheme proposed in the National Institute of Standards and Technology (NIST) standardization process for additional digital signature schemes [35, 36]. It is one of the nine candidate algorithms that advanced to the third evaluation round.

2.5.1 Key structure and trapdoor

The lattices in HAWK are based on the instantiation of NTRU lattices. The secret key consists of the basis matrix $\mathbf{B} = \begin{bmatrix} f & F \\ g & G \end{bmatrix}$, whose polynomials satisfy the NTRU equation $fG - gF = 1$. The public key in HAWK is the Gram matrix $\mathbf{Q} = \mathbf{B}^*\mathbf{B}$, where $\mathbf{B}^*$ is the Hermitian adjoint. The Gram matrix $\mathbf{Q}$ gives information about the length and the angles between the vectors of the basis without telling how they are oriented in space.

The trapdoor function in this scheme is the ability of the signer to sample short vectors in coset of the defined lattice with an offset based on the message, the verifier, having only $\mathbf{Q}$, can only measure the length (norm) of the signature vectors. Hence, the hash in hash-and-sign paradigm is the transformation of the message in a coset offset.

2.5.2 Security assumptions

The security behind the secret base recovery is given by the search module Lattice Isomorphism Problem. The Lattice Isomorphism Problem consists in determining whether two different lattices are isometric, that is, whether they are related via rotation and integer basis change. The search-module-LIP variant operates over the ring of integer polynomials $\mathcal{R}_n = \mathbb{Z}[x]/(x^n + 1)$.

The security behind the unforgeability of the signature is given by the one-more Shortest Vector Problem (omSVP). The omSVP is a variant of the Shortest Vector Problem (SVP) in which an adversary has access to an oracle that provides any number of short vectors; the problem states that it is not feasible for the adversary to produce a new short vector that is not trivially derived from the previously observed ones, given only the matrix $\mathbf{Q}$ and those vectors.

The omSVP assumption is recent and specific to HAWK. During the second round of the NIST evaluation, cryptanalysis found a flaw in the original definition of the problem: the problem, as first written, could be solved with an efficient computation. The flaw did not lead to practical forgeries against the scheme [36]. The version 1.1 specification corrected the definition to exclude this case; the security reduction still holds with essentially the same loss [35]. NIST considers

the short cryptanalytic history of omSVP and smLIP the main open question for HAWK in the third round [36].

2.5.3 Comparison with Falcon

HAWK shares core similarities with Falcon but design differences make it a valid alternative. First, the signatures are slightly smaller, at the cost of a larger public key. Second, signature generation executes entirely with integer arithmetic removing floating-point requirements and allowing implementation on devices with weak or absent Floating-Point Units (FPUs). Third, the algorithm eliminates rejection sampling, although statistical limits still require restarts during key generation and signing. Finally, while Falcon opted for true randomization, HAWK opted for derandomization, hence both the salt applied to the message salt and the sampling seed are generated deterministically. This approach allows the execution of the signature scheme on devices that lack a True Random Number Generator (TRNG) or rely on an insecure RNG.

2.5.4 HAWK operations

The HAWK specification document presents two parameter sets named HAWK-512 and HAWK-1024. HAWK-512 reaches NIST security level 1 and HAWK-1024 reaches NIST security level 5. The principal elements in HAWK are polynomials of degree n-1 in the ring $\mathcal{R}_n = \mathbb{Z}[x]/(x^n + 1)$ with n=512 for HAWK-512 and n=1024 for HAWK-1024.

Unlike ML-DSA and Falcon, HAWK represents polynomial coefficients as integers in $\mathcal{R}_n = \mathbb{Z}[x]/(x^n + 1)$ instead of modulo a fixed prime. Its Number Theoretic Transform is therefore computed modulo a single small prime during signing, where the operands are small, and modulo two distinct larger primes during verification, where the Q-norm produces integer values too large for one small prime to represent without wrap-around.

Key pair generation

The private key in HAWK is the matrix $\mathbf{B}$ that generates a lattice $\mathbb{Z}^{2n}$, defined as $\mathbf{B} = \begin{pmatrix} f & F \\ g & G \end{pmatrix}$ and its inverse as $\mathbf{B}^{-1} = \begin{pmatrix} G & -F \\ -g & f \end{pmatrix}$.

Key pair generation begins by sampling the coefficients of the polynomials f and g from a centered binomial distribution, using a pseudorandom stream expanded from an initial seed. These polynomials must satisfy specific norm constraints: $||(f, g)||_\infty \leq n/128$ and $||(f, g)||^2 > 2n\sigma^2_{krsec}$. Once f and g are validated, the algorithm computes the polynomials F and G. These computed polynomials must satisfy both the bound $||(F, G)||_\infty \leq 127$ and the NTRU equation $fG - gF = 1$

Algorithm 7 HAWK Key Pair Generation

Output: Secret key $sk = (kgseed, F \bmod 2, G \bmod 2, hpub)$, public key $pk = (q_{00}, q_{01})$

repeat
  $kgseed \leftarrow \mathbb{B}^{kgseedlen}$ ▷ Sample the key generation seed
  $(f, g) \leftarrow \text{Regeneratefg}(kgseed)$ ▷ Centred binomial sampling
  $(F, G) \leftarrow \text{NTRUSolve}(f, g)$ ▷ Solve $fG - gF = 1$
  $\mathbf{B} \leftarrow \begin{bmatrix} f & F \\ g & G \end{bmatrix}$ ▷ Secret basis, unimodular by construction
  $\mathbf{Q} \leftarrow \mathbf{B}^{\star}\mathbf{B}$ ▷ Public Gram matrix, self-adjoint
until f, g invertible mod 2, $\|(f, g)\|^2 > 2n\sigma_{\text{krsec}}^2$, q_{00} invertible, $(F, G) \neq \perp$ with $\|(F, G)\|_\infty \leq 127$, and q_{11} within bounds
$pk \leftarrow (q_{00}, q_{01})$ ▷ First row of $\mathbf{Q}$; the rest follows from $q_{00}q_{11} - q_{01}q_{10} = 1$
$hpub \leftarrow \text{SHAKE256}(pk)$ ▷ Bind the public key into the private key
return $sk = (kgseed, F \bmod 2, G \bmod 2, hpub)$, $pk = (q_{00}, q_{01})$

(mod X^n+1). The public key is the Gram matrix $\mathbf{Q}$, structured as $\begin{pmatrix} q_{00} & q_{01} \\ q_{10} & q_{11} \end{pmatrix}$. The matrix $\mathbf{Q}$ is self-adjoint, which establishes the relations $q_{00} = q_{00}^*$, $q_{11} = q_{11}^*$, and $q_{10} = q_{01}^*$. The NTRU equation also holds for matrix $\mathbf{Q}$, hence $q_{00}q_{11} - q_{01}q_{10} = 1$. A verifier can reconstruct the matrix $\mathbf{Q}$ knowing only the polynomials q_{00} and q_{01}, for this reason the public key bundle only contains those two polynomials. The secret key bundle contains only the parity of polynomials F and G to save memory, the seed used to generate polynomials f and g and *hpub*. If the algorithm fails to verify any of the constraints, the generation process restarts from the beginning with a new seed.

Similar to Falcon, HAWK's key generation in the reference implementation uses a fail-early approach, performing several checks on the sampled polynomials f and g before running the NTRU solver:

Parity check: the implementation evaluates the parity of f and g by summing their coefficients modulo 2. Both sums must be odd to ensure that f and g are invertible modulo 2 in $\mathbb{Z}_2[x]/(x^n + 1)$.

Norm lower bound: it verifies that $\|(f, g)\|^2 > 2n\sigma_{\text{krsec}}^2$, immediately discarding pairs where (f, g) is too short.

Invertibility of q_{00} modulo p_1 and p_2: to guarantee that $q_{00} = f^\star f + g^\star g$ is invertible, the code transforms f and g into the NTT domain modulo primes p_1 and p_2. It computes $\hat{q}_{00} \pmod{p_1}$ and checks that no coefficients are zero, also in modulo p_2.

Bound on $1/q_{00}$: it checks that the constant coefficient (degree 0) of $1/q_{00}$ is strictly smaller than the threshold β_0. This is evaluated by converting q_{00} to fixed-point representation, computing its FFT, inverting it in the FFT domain, and returning to the coefficient domain via inverse FFT.

Only if all the checks pass, `solve_NTRU` is called to solve $fG - gF = 1$ and find F and G. If the solver succeeds, next it is checked that the infinity norm satisfies $\|(F, G)\|_\infty \leq 127$ and validates the coefficient bounds of the Gram matrix components q_{00}, q_{01}, and q_{11}; if any bound is violated, the loop restarts with a fresh seed.

Signature generation

In signature generation, the public key hash $hpub = hash(q_{00}\|q_{01})$ is precomputed. The message representative M is then computed as $hash(m\|hpub)$. The algorithm concatenates M with a random salt and hashes the result to generate a binary target vector $h = (h_0, h_1)$, where h_0 and h_1 belong to $\mathcal{R}_n/2\mathcal{R}_n$, i.e., their coefficients are 0 or 1. The vector h defines the target coset $2\mathcal{R}_n^2 + h$ in lattice coordinates. To perform sampling in absolute coordinates, the algorithm calculates a standard coordinate offset $t = B \cdot h \pmod 2$, which defines the sampling coset $2\mathbb{Z}^{2n} + t$.

The signer samples a vector x from a discrete Gaussian distribution over the shifted lattice $2\mathbb{Z}^{2n}+t$ using the standard deviation parameter $2\sigma_{sign}$ If the Euclidean norm of x exceeds the limit defined by the parameter set, the algorithm rejects the vector and restarts the sampling process with a new salt. The valid vector x represents a lattice point in absolute coordinates.

The algorithm translates this point back into lattice coordinates by computing $w = B^{-1}x$. Because the secret basis matrix B is unimodular, the mapped vector w strictly belongs to the target coset $2\mathcal{R}_n^2 + h$ [35]. Finally, the algorithm computes the signature vector $s = (h - w)/2$ that is sure to have integer coefficient due to the fact that w and h have the same parity. The signature bundle is the tuple containing the salt and the polynomial component s_1 of vector s.

In the source code, the computation of $\mathbf{t} \leftarrow \mathbf{Bh} \pmod 2$ is done in the standard coefficient domain with bitwise operations. Due to $\mathbf{h}$ being binary (with coefficients in $\{0, 1\}$), the multiplication uses F mod 2 and G mod 2 directly from the secret key, while f and g are reduced modulo 2 in place. At the end of the procedure, the signature will only contain s_1 without s_0, which would be redundant for the verifier. Instead the verifier will recompute s_0 as part of the verification process. This brings a simplification in signing: the procedure only needs to compute w_1, avoiding the evaluation of w_0. Since $w_1 = fx_1 - gx_0$, it depends strictly on f, g, and the sampled vector $\mathbf{x}$ from the discrete Gaussian, avoiding the need to reconstruct the full polynomials F and G, whose only parity was saved. The polynomial multiplications required to evaluate w_1 are performed in the NTT domain modulo the 15 bits prime 18433; because the coefficients are small, this modular arithmetic matches the exact integer polynomial product without overflow. The only condition that triggers a restart in the signature procedure is if the squared norm of the sampled vector exceeds the verification bound ($\|\mathbf{x}\|^2 > 8n\sigma^2_{\text{verify}}$). The specification shows that this is extremely rare, with a restart probability of about 2^{-17} for HAWK-512 and 2^{-121} for HAWK-1024.

Algorithm 8 HAWK Signature Generation

Input: Message m, secret key $sk = (kgseed, F \bmod 2, G \bmod 2, hpub)$
Output: Signature $sig = (salt, s_1)$

$(f, g) \leftarrow$ Regeneratefg($kgseed$) ▷ Regenerate the secret basis
$M \leftarrow$ SHAKE256($m \,\|\, hpub$) ▷ Message representative
$a \leftarrow 0$ ▷ Initialize attempt counter
repeat
  $salt \leftarrow$ SHAKE256($M \,\|\, kgseed \,\|\, a \,\|\,$ Rnd($saltlen$)) ▷ Derandomized salt
  $\mathbf{h} \leftarrow$ SHAKE256($M \,\|\, salt$) ▷ Binary target vector
  $\mathbf{t} \leftarrow \mathbf{Bh} \bmod 2$ ▷ Sampling coset in standard coordinates
  $seed \leftarrow M \,\|\, kgseed \,\|\, a{+}1 \,\|\,$ Rnd(320) ▷ Sampler seed
  $\mathbf{x} \leftarrow D_{2\mathbb{Z}^{2n}+\mathbf{t},\, 2\sigma_{\text{sign}},\, \mathbf{0}}(seed)$ ▷ Drawn by SamplerSign()
  $a \leftarrow a + 2$ ▷ Advance the counter
  if $\|\mathbf{x}\|^2 > 8n\sigma^2_{\text{verify}}$ **then**
    continue ▷ Reject and restart with a new salt
  end if
  $w_1 \leftarrow fx_1 - gx_0$ ▷ Second component in lattice coordinates
  if SymBreak(w_1) = *false* **then**
    $w_1 \leftarrow -w_1$ ▷ Fix the sign convention
  end if
  $s_1 \leftarrow (h_1 - w_1)/2$ ▷ Signature polynomial
until s_1 encodes within the fixed signature size
return $sig = (salt, s_1)$

Signature Verification

To verify the signature, the verifier first concatenates the message m with $hpub = hash(q00||q01)$ and hashes the data to reconstruct the message representative M that will concatenate with the salt in the signature bundle to reconstruct the vector $h = (h_0, h_1)$ as $h = hash(M||salt)$. The algorithm uses the received signature polynomial s_1 and the public key to compute the missing polynomial s_0. This reconstruction applies the following relation [35]:

$$s_0 = \left\lfloor \frac{h_0}{2} + \frac{q_{01}}{q_{00}} \left(\frac{h_1}{2} - s_1 \right) \right\rceil .$$

Once s_0 is reconstructed, the verifier can compute the vector $w = h - 2s$. The signature is validated if the Q-norm squared of w satisfies the threshold $||w||_Q^2 \leq 4 \cdot 2n \cdot \sigma_{verify}^2$.

In the verification procedure of the source code, the verifier first recomputes $w_1 = h_1 - 2s_1$ with h_1 derived from the message hash and salt and s_1 received in the signature bundle. To reconstruct the missing component s_0, the specification formula $s_0 = \lfloor \frac{h_0}{2} + \frac{q_{01}}{q_{00}}(\frac{h_1}{2} - s_1) \rceil$ is rewritten by noting that $\frac{h_1}{2} - s_1 = \frac{w_1}{2}$. Factoring out $\frac{1}{2}$, the relation simplifies to:

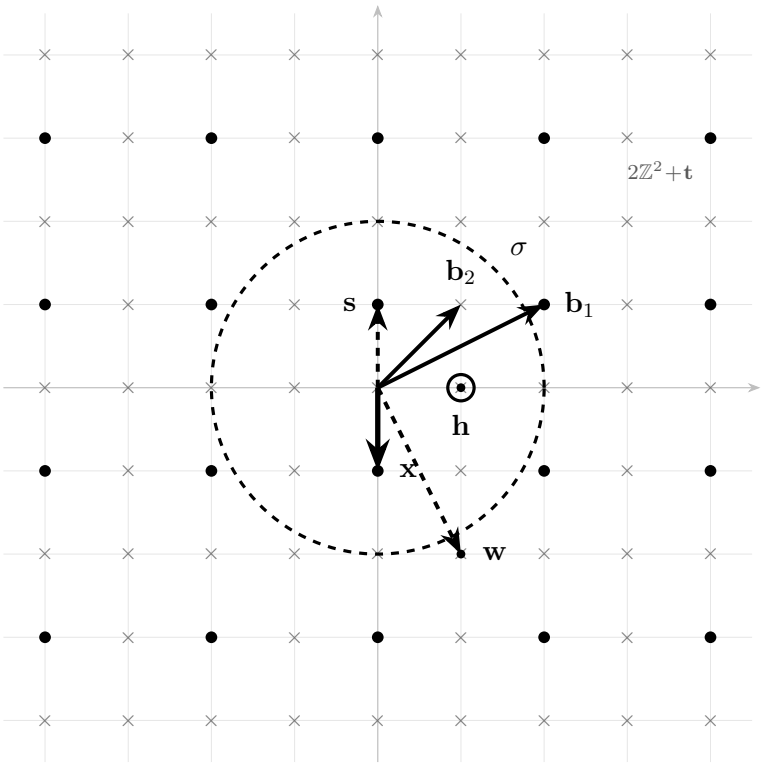
$$s_0 = \left\lfloor \frac{1}{2} \left(h_0 + \frac{q_{01} w_1}{q_{00}} \right) \right\rceil .$$

The term $(q_{01}w_1)/q_{00}$ is computed by converting the polynomials into a 32 bits fixed point representation and transforming them into the frequency domain via FFT. This allows the polynomial multiplication and division by q_{00} to be evaluated pointwise. After returning to the coefficient domain via `fx32_iFFT`, the result is added to h_0 (also in fixed-point format) and rounded to the nearest integer to obtain s_0. Finally, the first lattice coordinate is recovered as $w_0 = h_0 - 2s_0$. The verifier computes its quadratic norm under $\mathbf{Q}$ through an algebraic reformulation that eliminates the need for the missing component q_{11}, expressing the norm using only q_{00} and q_{01}.

Algorithm 9 HAWK Signature Verification

Input: Message m, public key $pk = (q_{00}, q_{01})$, signature $sig = (salt, s_1)$
Output: Boolean

$hpub \leftarrow \text{SHAKE256}(pk)$ ▷ Recompute the public key hash
$M \leftarrow \text{SHAKE256}(m \,\|\, hpub)$ ▷ Message representative
$(h_0, h_1) \leftarrow \text{SHAKE256}(M \,\|\, salt)$ ▷ Rebuild the binary target vector
$w_1 \leftarrow h_1 - 2s_1$ ▷ Recover the second component
if $\text{SymBreak}(w_1) = false$ **then**
    return false ▷ Wrong sign convention
end if
$w_0 \leftarrow \text{RebuildS0}(q_{00}, q_{01}, w_1, h_0)$ ▷ Reconstruct the missing component
if $w_0 = \perp$ **then**
    return false ▷ Reconstruction out of range
end if
$r \leftarrow \|\mathbf{w}\|^2_{\mathbf{Q}}$ ▷ $\mathbf{Q}$-norm of $\mathbf{w} = (w_0, w_1)$
return $r \leq 8n\sigma^2_{\text{verify}}$ ▷ Accept only if the vector is short



$\mathbf{b}_1 = (2,1), \quad \mathbf{b}_2 = (1,1)$	secret basis (columns of B)
$\mathbf{h} = (1,0)$	message hash, binary coordinates
$\mathbf{t} = B\mathbf{h} \bmod 2 = (0,1)$	active coset index (black dots)
$\mathbf{x} = (0,-1)$	sampled from $2\mathbb{Z}^2 + \mathbf{t}$, Euclidean coordinates
$\mathbf{w} = B^{-1}\mathbf{x} = (1,-2)$	B-lattice coordinates of $\mathbf{x}$; $\mathbf{w} \equiv \mathbf{h} \pmod 2$
$\mathbf{s} = (\mathbf{h} - \mathbf{w})/2 = (0,1)$	transmitted signature

Figure 2.2: Two-dimensional toy representation of the HAWK signature scheme geometry.

Chapter 3

Bootloader Rationale and Implementation

This chapter is composed of three parts. The first describes the bootloader as a class of application and which design choices separate one bootloader from another. The second describes the target hardware, the Infineon AURIX TC375TP System on a Chip (SoC), including the TriCore processor and the memory layout. The third describes the secure bootloader implemented in this work, its dual-slot firmware update mechanism, its boot decision steps and its cryptographic verification process.

3.1 Bootloader Fundamentals

A microcontroller begins execution from a fixed address after every reset. The code at that address is placed by the manufacturer. This initial firmware is typically referred to as the BootROM. Depending on the device, the BootROM parameters may be configurable, but the underlying code is not replaceable. Carrying the device from that state to the first instruction of the application is the task of the bootloader.

RFC 9019 defines the component by when it runs and by what it decides: a bootloader is a piece of software executed once a microcontroller has been reset, and it is responsible for deciding what code to execute [2].

Device manufacturers describe the same component as the initial code that runs after power-up, whose task is to bring up the next boot stage, and which may also provide low-level software update and support secure boot [41].

A software bootloader is itself an application. It has its own linker script, manages the startup code and its own memory regions. In particular in flash memory that the application cannot use, so the bootloader has a cost that is paid on every device of the production run; Section 3.3.2 reports it for the implementation

of this work.

The startup code prepares the runtime environment of one program: it sets the stack pointer, copies initialized data into RAM and clears the zero-initialized sections. The bootloader operates one level above: it chooses which program the startup code will prepare.

The handover is a jump without return. Nothing restores the machine state afterwards, so the bootloader must leave the hardware as the application expects to find it.

3.1.1 Bootloader Rationale

A device can be programmed with the application alone and start it directly at reset. A separate bootloader exists because three requirements are not satisfied by that arrangement.

Updatability and recovery. Firmware must be replaceable after the device has left the factory, without a debug probe and without production equipment. The bootloader holds the ability to erase and write program flash, so the application does not need it. Replacement alone is not sufficient: an image can be defective, and an update can be interrupted. RFC 9019 states that a device recovering from invalid firmware has two options, and that a bootloader must implement one of them, either selecting a different valid image already present or obtaining a new one [2].

The first option requires the device to keep more than one image, and therefore a rule that decides between them and refuses an image older than the one already accepted. The second option requires a mode in which the bootloader receives an image instead of starting one.

Separation of concerns. Hardware bring-up, image selection and image installation are performed once, in a component that does not change when the application changes.

Authenticity. Boot time is the only point in the lifecycle in which an image can be checked before it is executed. This requirement is the one that cannot be delegated to the application itself. Software cannot establish its own integrity, because a modified image can be modified further to report a successful verification [42]. For this reason, a component that runs before the image, and that the image cannot alter, is required.

3.1.2 Structure and Execution Flow

Boot stages. The code executed after reset is placed at a fixed address by the silicon vendor. It brings the device out of the reset state and then hands control to

a second stage held in flash memory. The first stage cannot be corrected after the deployment: a defect in it is repaired only by replacing the hardware [41].

Two properties are required of that first stage. An attacker must not be able to change the reset vector, and the code it points to must be protected [41].

Figure 3.1 shows the resulting arrangement. The number of software stages is not fixed. It grows with the distance between the memory that holds the application image and the memory the processor can execute from at reset. On a microcontroller the application image is mapped in flash memory and is executed in place.

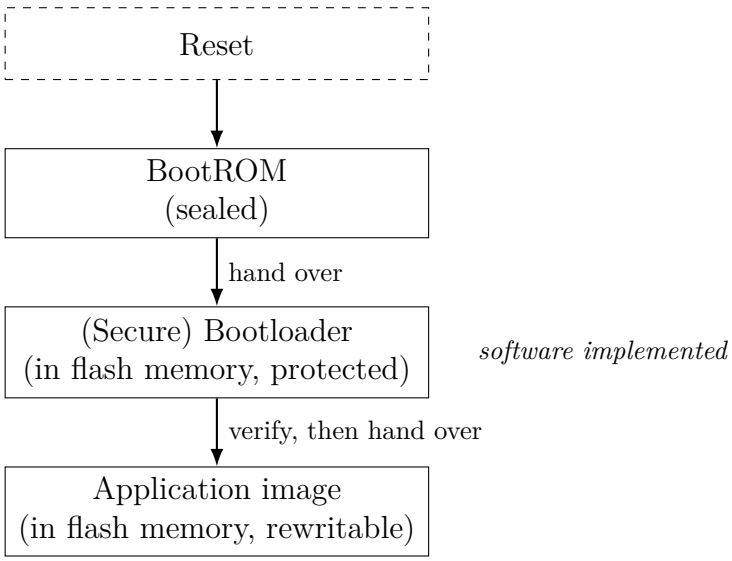


Figure 3.1: Boot stages on a microcontroller. The BootROM is sealed at manufacturing time and cannot be corrected in the field. The bootloader is the first stage implemented in software. A single software stage is sufficient when the application image is executed in place from flash memory.

Order of work. The bootloader performs its work in a fixed order. It first brings up the part of the hardware that its own execution requires. It then reads the metadata of the images available in flash memory and selects a candidate. It verifies the candidate. It finally hands control to the entry point of the selected image. A device that supports field updates adds one further stage, in which the bootloader receives and stores a new image before the selection takes place.

Image layout. An image is not only executable code. The bootloader reads metadata stored alongside it before it decides anything, and it can do so only if the position and the format of that metadata are agreed in advance [2].

The layout is therefore part of the contract between the tool that produces the image and the bootloader that consumes it.

Update mode. A bootloader that implements the second recovery option needs a defined way of being told to run an update routine instead of starting the application. The common triggers are an external signal on a general-purpose pin, a request left in memory by the application before a reset or the absence of any valid image.

3.1.3 Secure Bootloader

A secure bootloader is a bootloader with one added responsibility: before it hands control to an image, it checks that the image is the one the manufacturer released, and it has a defined behaviour when the check fails. The mechanisms available for that check are described in Section 3.1.4. NIST SP 800-193 frames the resulting requirements as protection, detection, and recovery of platform firmware [43].

3.1.4 Bootloader Design Classes

The bootloader described in Section 3.3 is a software implementation. It verifies one application image with an asymmetric signature, monolithically and in sequential mode, on the first application core, and it holds the trust anchor inside its own image rather than in a hardware-protected region. The consequences of the last choice are stated in Section 3.3.7 and in Section 5.3.

Bootloaders that satisfy the same functionalities still differ in how they are implemented, and the differences fall into a number of design choices.

The first distinction is the device itself. RFC 9019 groups IoT devices into five types according to their processor and memory architecture: a single microcontroller executing code in place from its own flash memory; a single processor partitioned into a secure and a normal mode, with one image for each; a symmetric multiprocessor running a general purpose operating system, commonly from external storage and not in place; and two arrangements of heterogeneous processors, one sharing flash and RAM and one keeping separate memories connected by a communication channel [2]. The types after the first move the choices below onto different hardware, and in the partitioned and in the heterogeneous cases they make the update of one image depend on the update of another. The device of this work matches the first type: a single microcontroller, code executed in place, one image to select and verify.

Start of the root of trust. The first stage is trusted without being verified, so what makes it trustworthy is that it cannot be rewritten. The common solutions are a mask ROM fixed at manufacturing, a one-time-programmable flash region that has been programmed and locked, and a dedicated security core with private memory [41]. The choice fixes which region an attacker with write access to flash cannot reach.

Processor that performs the verification. The verification can run on the same core that later executes the application, or on a separate core reserved for security functions. Renesas describes both arrangements on the same microcontroller family. In the first, the device has no security-only processor and the secure boot logic is executed by the application core from one-time-programmable memory. In the second, a separate cryptographic processor executes the security software and releases the other cores from reset only after the verification has succeeded [44]. The second arrangement removes the verification from the application core at the cost of a dedicated processor and an inter-processor interface.

Order between verification and execution. Sanwald et al. classify three execution modes. In sequential mode the authentication of a stage completes before control passes to it. In concurrent mode the current stage runs while the next one is authenticated, which requires at least two independent processing units. In parallel mode the next stage starts before its authentication has finished, so it belongs to authentic boot rather than to secure boot [1].

Renesas describes the overlap between verification and execution as running the two in parallel, and reports it as typical of multicore devices [41]; Sanwald et al. reserve that word for the mode in which execution starts first.

Integrity mechanism. Three mechanisms are used to decide whether an image is acceptable. A checksum or a cyclic redundancy check detects accidental corruption and provides no protection against a deliberate modification, since the value can be recomputed by anyone. A message authentication code is keyed and fast, but the same secret key is present on the device and at the signer, so it must be held in protected storage, and the mechanism does not provide non-repudiation by default [41].

A digital signature keeps the private key outside the device. What the device must protect is no longer a secret, but the integrity of a public value: the public key, or a hash of it, must be impossible to replace [41]. An attacker who extracts the full content of one device therefore obtains nothing that another device would accept.

3.2 AURIX Development Board

This work is based on a Infineon AURIX TC375TP system on a chip (SoC). The Infineon AURIX TC3xx is a family of 32-bit microcontrollers oriented for automotive environment. The family provides SoCs up to 6 TriCore 1.6.2P CPUs up to 300 MHz and developed with hardware safety mechanisms suitable for domains like Anti-lock braking systems (ABS), airbag deployment, powertrain control, Advanced Driver-Assistance Systems (ADAS) and even infotainment. The architecture meets ISO 26262 ASIL-D and IEC 61508 SIL-3 standards for functional safety.

The TriCore architecture integrates RISC (Reduced Instruction Set Computer) little-endian processor, a Digital Signal Processor (DSP) capability, and a microcontroller unit for sensors and peripherals. Each core contains a dedicated floating-point unit (FPU) for single-precision (32-bit) arithmetic and supports DSP-oriented instructions for the parallel execution of 16-bit and 8-bit integer operations. The architecture implements CPU watchdogs and optional hardware lockstep mechanisms to mitigate hardware failures via physical isolation and two clock cycle execution delay between paired cores.

Every core contains 16 32-bit Data Registers (D0 to D15) and 16 32-bit Address Registers (A0 to A15). The architecture does not implement dedicated floating-point registers; instead, floating-point operations make use of the standard data registers.

Some of the address registers have explicit roles during execution: register A15 is the implicit base address register, A11 stores the return address for calls and interrupts, and A10 acts as the Stack Pointer (SP). The register set is divided in upper and lower context; the upper context registers are saved in function calls, interrupt and trap routines while the lower context is managed manually.

The TriCore memory subsystem implements a 32-bit address space partitioned into 16 segments of 256 MB each; the most significant hexadecimal digit of the address selects the target segment. Non-volatile storage consists of Program Flash (PFLASH) and Data Flash (DFLASH) banks, both protected by hardware ECC. The Data Memory Unit (DMU) arbitrates command sequencing for both flash types.

3.2.1 Hardware Security Module

The SoC in the TC3xx family integrates a Hardware Security Module (HSM) based on a 32-bit ARM Cortex-M3 core operating at 100 MHz. The HSM provides symmetric and asymmetric cryptographic accelerators, supporting AES-128, True Random Number Generation (TRNG), and hash functions including MD5, SHA-1, and SHA-256. The hash engine processes 512-bit blocks in 65 clock cycles. The asymmetric accelerator supports Elliptic Curve Cryptography (ECC) operations. For a 256-bit curve, it achieves 200 ECDSA signatures per second and 100 verifications per second at 100 MHz, which corresponds to approximately 5 ms per signature and 10 ms per verification. The HSM is only available via commercially licensed compilers, vendor-specific libraries or closed firmware stacks; for this reason a Deterministic Random Byte Generator (DRBG) based on seeded SHAKE instance is used in place of the TRNG.

This HSM presents architectural limitations for Post-Quantum Cryptography (PQC). The hardware lacks dedicated accelerators for the polynomial multiplication and Keccak (SHA-3/SHAKE) operations required by post-quantum signature

schemes. Consequently, these algorithms are implemented in software and run on the main cores of the SoC.

The ARM Cortex-M3 processor implements variable-cycle long multiplication instructions (e.g., smull and smlal). Abdulrahman et al. observe that these operations are not executed in constant time and, consequently, cannot be safely used when handling secret data [45]. Similarly, Greconici et al. state that the Cortex-M3 does not have a constant-cycle 32-bit multiplier producing a 64-bit result, which causes software implementations to leak timing information about secret integers [46]. These hardware constraints make the TC3xx HSM unfavorable for timing side-channels when executing software-based PQC operations.

Software execution on the primary cores also preserves cryptographic agility: as PQC standardization remains an active process, a software stack accommodates future algorithmic revisions without hardware modification.

3.2.2 Development Board and Environment

The board used for the development is the SAK-TC375TP-96F300W AA, containing three TriCore CPUs operating at 300 MHz, 6 MiB of Program Flash memory composed of two 3 MiB banks and two Data Flash banks of 256 KiB and 128 KiB. The PCB includes an on-board miniWiggler for USB debugging, an external power supply circuit, an Infineon CAN transceiver, and an Ethernet physical layer transceiver. It incorporates general-purpose user buttons, LEDs, and an analog potentiometer connected to the Analog-to-Digital Converter (ADC).

The development environment is the AURIX Development Studio (ADS), an Eclipse-based IDE bundling build tools and the Infineon Low Level Driver (iLLD) library for bare-metal firmware development. The integrated debugger communicates over the Device Access Server (DAS) protocol, supporting hardware flashing, breakpoint configuration, and instruction stepping.

The TC3xx supports two toolchains: the open-source GNU Compiler Collection (GCC) and the proprietary TASKING VX-toolset; this work uses the TASKING VX-toolset for AURIX Development Studio v1.1r8, non-commercial release. The TASKING compiler conforms to ISO C99 and the mandatory features of ISO C11. MISRA-C 2004 is a coding standard widely required for automotive software; the TASKING VX-toolset provides a MISRA-C checker, which was enabled during development, and the project code was aligned to its guidance in part. Memory layout and section placement are defined via the proprietary Linker Script Language (LSL), which maps executable sections and data segments to absolute addresses across the physical memory space.

Table 3.1 details the TriCore register file and context classification, showing address registers ($\mathbf{A}_0$–$\mathbf{A}_{15}$) and control pointers (left columns) with data registers ($\mathbf{D}_0$–$\mathbf{D}_{15}$) and status registers (right columns). This registers' classification dictates

context preservation during execution: the hardware automatically saves and restores the upper context (comprising registers $\mathbf{A}_{10}$–$\mathbf{A}_{15}$, $\mathbf{D}_{8}$–$\mathbf{D}_{15}$, **PCXI**, and **PSW**) on interrupts and function calls, whereas lower context registers ($\mathbf{A}_{2}$–$\mathbf{A}_{7}$ and $\mathbf{D}_{0}$–$\mathbf{D}_{7}$) are preserved under software control.

Table 3.1: TriCore register map and context classification.

Address & System Registers		**Data & System Registers**	
Register	**Usage / Classification**	**Register**	**Usage / Classification**
FCX	Free Ctx Head Pointer (Sys)	LCX	Free Ctx Tail Pointer (Sys)
ISP	Interrupt Stack Pointer (Sys)	ICR	Interrupt Control Reg (Sys)
PIPN	Pending Interrupt No. (Sys)	BIV	Base Interrupt Vector Ptr (Sys)
BTV	Base Trap Vector Ptr (Sys)		
PC	Program Counter (Sys)		
PCXI	Prev. Context Reg (Up/Low)	PSW	Program Status Word (Upper)
A15	Implicit base address (Upper)	D15	Implicit data register (Upper)
A14	General-purpose address (Upper)	D14	General-purpose data (Upper)
A13	General-purpose address (Upper)	D13	General-purpose data (Upper)
A12	General-purpose address (Upper)	D12	General-purpose data (Upper)
A11	Return Address (Up/Low)	D11	General-purpose data (Upper)
A10	Stack Pointer (Upper)	D10	General-purpose data (Upper)
A9	Global register	D9	General-purpose data (Upper)
A8	Global register	D8	General-purpose data (Upper)
A7	Function Argument 4 (Lower)	D7	Function Argument 4 (Lower)
A6	Function Argument 3 (Lower)	D6	Function Argument 3 (Lower)
A5	Function Argument 2 (Lower)	D5	Function Argument 2 (Lower)
A4	Function Argument 1 (Lower)	D4	Function Argument 1 (Lower)
A3	Return Value (Lower)	D3	Return Value (Lower)
A2	Return Value (Lower)	D2	Return Value (Lower)
A1	Global register	D1	General-purpose data (Lower)
A0	Global register	D0	General-purpose data (Lower)

3.3 Bootloader Architecture and Design

Considering the BootROM of AURIX as the first stage bootloader, this secure bootloader constitutes the second stage in the platform chain of trust on the Infineon AURIX TC375TP. The factory-programmed BootROM hands control to the custom second-stage bootloader, which authenticates one of two firmware images stored in program flash before initiating application execution. The design implements signature verification using both classical and quantum-resistant signature schemes and a dual-slot firmware management policy with a copy-on-update mechanism and an Ethernet-based firmware upload mechanism built on the lwIP stack [24].

3.3.1 System Architecture

This section describes the architectural design of the bootloader: its three firmware security properties, the staged verification pipeline, the dual-slot partitioning, and the root-of-trust mechanism.

The bootloader architecture is characterized by three firmware security properties: hardware execution mode, software integrity scheme, and platform resiliency.

First, despite the multicore architecture of the AURIX TC375TP, the boot sequence operates in sequential execution mode [1], where the bootloader completes authentication before transferring execution control.

Second, the software integrity scheme relies on signature-based authentication [1], integrating both classical and post-quantum cryptographic algorithms. The hardware trust anchor is designated for One-Time-Programmable (OTP) memory in production [1], though the current development iteration simulates this Root of Trust for Storage (RTS) [47] within the bootloader firmware image as a static array.

Third, the architecture targets the "Protected" platform resiliency property specified in NIST SP 800-193 [43]. The bootloader acts as the Root of Trust for Detection (RTD) [43]: it verifies firmware integrity via digital signature prior to execution. The boot decision logic acts on that result and halts the boot sequence upon authentication failure, so no unauthenticated image is executed. The same document also requires the regions holding the roots of trust to be immutable or write-protected; the prototype assumes this of the bootloader region rather than enforcing it in hardware (Section 5.3).

Software Update framework RFC 9019 [2] defines a manifest-centric architecture for firmware updates on constrained embedded devices. Each firmware package includes a signed manifest containing the cryptographic digest, version sequence number, target platform identifier, and installation conditions. The proposed bootloader architecture adopts this separation between metadata and firmware binary, and stages its verification accordingly.

Verification strategy. During initialization, the bootloader eagerly evaluates structural metadata for every populated slot: magic byte validation, algorithm and hash identifier bounds, size constraints, and top-to-bottom header consistency. Cryptographic signature verification is invoked only for slots that pass all structural checks and satisfy the version threshold, avoiding the overhead of hashing the full firmware binary on structurally invalid images.

Chain of trust. The first-stage ROM bootloader, embedded during silicon manufacturing, is the Root of Trust for Measurement (RTM) [47]: the first module in the bootstrap chain, realized and protected by hardware. Following Boot Mode Header Descriptor (BMHD) validation, it transfers execution to the custom second-stage bootloader in program flash, which authenticates the application image via signature verification. Authentication failure halts the boot sequence; the bootloader implements the secure boot variant [47], not authenticated boot, where the system records the violation and continues.

Execution model. The bootloader logic executes exclusively on CPU0 within a single-core context. The secondary processing units, CPU1 and CPU2, remain in a halted state until the firmware candidate is fully authenticated and the execution transfer is triggered. This is how the sequential execution mode stated above is realized on the target. Furthermore, constraining the initialization sequence to a single master core ensures deterministic startup behavior and avoids concurrency risks, consistent with the design principles specified in ISO 26262-6:2018 [48].

Taken together, the single-master-core initialization, the staged structural filtering performed before signature verification and the late release of the application cores satisfy the non-bypassability and protection guidelines for critical platform firmware defined in NIST SP 800-193, Section 4.2.3 [43].

Slot partitioning. Two firmware slots reside in program flash with different roles. Slot A is the exclusive execution slot; the application always launches from Slot A. Slot B serves as the staging area for incoming firmware. When Slot B wins the boot election, its contents are copied into Slot A before execution begins. The TC375TP provides a hardware flash bank swap mechanism based on virtual addressing, but that mechanism requires writing configuration registers and asserting a device reset to take effect [9]. The actual architecture implements a software-driven copy-on-update policy rather than utilizing the hardware swap mechanism. While a hardware swap incurs less overhead, it obscures the validation state across reboots. A software-driven approach maintains explicit state control: if both the primary and backup slots fail validation, the bootloader deterministically drops into a recovery state to allow the upload of a new firmware image, preventing infinite reboot loops. It eliminates runtime branching on the executing slot identity and fixes the application code region at the Slot A base address independent of update history.

Root of trust. The proposed architecture embeds the trust anchor directly

in the bootloader binary. The compiler places the SHA3-256 digest of the trusted public key as a `const uint8_t` array in the read-only `.rodata` section. At runtime, the bootloader computes the hash of the public key read from the firmware candidate header and compares it against the stored reference. Because the anchor is stored without hardware-enforced protection, the security limits of the verification scheme depend on the integrity of the bootloader image itself. An adversary capable of writing to the bootloader flash memory could replace the stored digest with the hash of an unauthorized public key, and bypassing the signature verification. Consequently, the implementation assumes the immutability of the bootloader flash region (Section 1.1).

A production-ready architecture typically establishes a hardware root of trust by storing and sealing the reference hash within One-Time-Programmable (OTP) memory. However, managing program flash sectors write protection requires executing specific hardware command sequences for erasing, programming, and sealing the configuration. Because this flash management procedure introduces operational overhead and risk of mis-management the present implementation relies on the software-embedded array, which allows key updates via straightforward source code recompilation.

The automotive Secure Hardware Extension (SHE) specification [49], an industry standard defining an isolated hardware module for secure key storage and cryptographic execution, relies upon a symmetric authentication scheme. While this MAC-based approach provides low computational latency, it mandates secure out-of-band symmetric key distribution and isolated hardware to maintain key confidentiality. The proposed architecture employs asymmetric post-quantum signatures to authenticate the firmware candidate. This asymmetric approach eliminates the symmetric key distribution requirement, exchanging hardware-enforced confidentiality for increased computational latency and a software-defined trust boundary.

3.3.2 Memory Layout and Slot Organization

The TC375TP program flash is partitioned into a bootloader region in the PFLASH0 bank and two equal application slots in the PFLASH1 bank. Table 3.2 lists the region boundaries in the non-cached (segment `0xA`) address space, which the bootloader uses exclusively for all flash reads and writes.

The bootloader occupies the PFLASH0 bank. The bootloader binary, including the Infineon Low Level Driver runtime and the startup code, uses approximately 250 KiB of code and data; the rest of the 3 MiB bank is unused. Its size depends on the number and type of verification algorithms supported. The two application slots occupy the PFLASH1 bank.

The 1.5 MiB slot size is a development-time layout choice, not an architectural

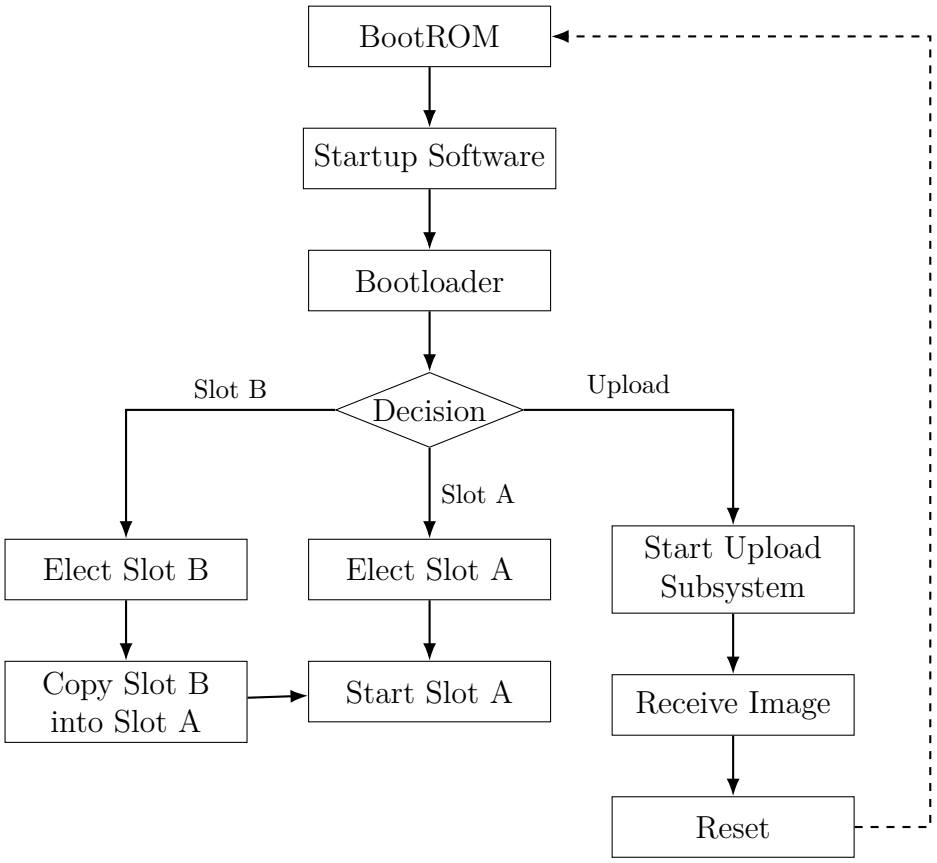


Figure 3.2: Bootloader system architecture. The ROM bootloader (first stage, immutable) transfers execution to the custom second-stage bootloader. The second-stage bootloader authenticates a firmware candidate in Slot A or Slot B and launches the application; if no valid candidate is found, the lwIP-based update subsystem activates to receive a new image via Ethernet into Slot B. The host-side signing tool `sign.py` produces the firmware image transmitted over TCP. The boot decision algorithm is described in Section 3.3.6.

Table 3.2: Bootloader and application regions in program flash (non-cached segment).

Region	**Base address**	**Size**	**Sectors**
Bootloader	`0xA0000000`	3 MiB	192 × 16 KiB
Slot A (primary)	`0xA0300000`	1.5 MiB	96 × 16 KiB
Slot B (secondary)	`0xA0480000`	1.5 MiB	96 × 16 KiB

limit. Reserving a small upper region of PFLASH0 for the bootloader only and assigning the remainder of each bank to one slot would provide approximately 2.5 MiB per slot while keeping each slot within a single flash bank at mirrored offsets. Realistic automotive images range from 1.1 to 5.9 MiB [50]; the upper end of this range is reached using external flash banks or by using larger members of the TC3xx family, which provide more program flash. The metadata header, the boot decision, and the verification path are independent of the slot size.

The AURIX architecture exposes the same physical flash at two address aliases: the non-cached segment (`0xA0xxxxxx`) and the cached segment (`0x80xxxxxx`).

The bootloader disables the program cache and addresses all flash reads through the non-cached segment to produce deterministic read latency and avoid coherency hazards.

Each slot begins with a fixed 8 KiB macro header block followed immediately by the application binary. The macro header holds the primary metadata header, the public key, and the digital signature, all 32-byte aligned, together with a copy of the metadata header at the final 96 bytes of the block.

3.3.3 AURIX Application Binary Code Organization

An application binary targeted at the TC375TP must satisfy alignment constraints imposed by the TriCore ISA for interrupt and trap vector tables. The standard code organization adopted in this design places these tables at the lowest addresses of the application binary region so that the alignment requirements are satisfied without additional padding.

Vector Table Alignment Requirements

The TriCore architecture defines two hardware-dispatched vector tables per core, alongside a designated startup vector region [9].

Interrupt Vector Table. Each core implements an 8 KiB Interrupt Vector Table (IVT) that must align to an 8 KiB boundary in memory. The IVT manages up to 255 discrete interrupt sources. The architecture binds each Interrupt Service Request to a unique Service Request Priority Number (SRPN); the interrupt number defines the priority level: 255 for the highest and 1 for the lowest, 0 is reserved. During initialization, the cores loads the IVT base address into the Base Interrupt Vector (BIV) register. The hardware dispatches incoming interrupt requests by adding a scaled priority offset to this base address.

3.3.4 Threat Model

The asset protected by the bootloader is the authenticity and integrity of the application firmware executed by the MCU, together with the trust anchor that

authorizes it. The adversary is assumed able to deliver a modified firmware image over the update channel and to read or write the application slot region of program flash (the secondary slot, Section 3.3.2), but unable to obtain the offline signing private key, unable to modify any other PFLASH region (the bootloader code and the trust anchor), and unable to modify the anti-rollback version counter held in DFLASH0 (Section 3.3.6). A production deployment would have to enforce these assumptions in hardware: by write-protecting the bootloader bank, by locking the debug interface so the device can no longer be debugged, by sealing the immutable trust anchor in One-Time-Programmable (OTP) memory, and by isolating the rewritable version counter from the application cores through the Memory Protection Unit, or through the HSM-exclusive DFLASH1 region where an HSM is available (Sections 3.3.7 and 5.3) [1, 51]. Within this model the bootloader must reject unsigned or modified firmware and reject downgrades to superseded versions. Side-channel and fault-injection attacks on the verification routine and compromise of the host signing key are out of scope for this prototype, and are revisited in the scope boundaries of Section 1.3.

Trap Vector Table. Each core implements a 256-byte Trap Vector Table (TVT) that must align to a 256-byte boundary. The TriCore specification defines eight general trap classes. During initialization, the core loads the TVT base address into the Base Trap Vector (BTV) register. The hardware dispatches traps by adding a scaled class offset to this base address.

Core Startup Vector. Each core requires a designated 32-byte memory region at a deterministic address. This region contains the primary jump instruction sequence responsible for transferring execution control from the hardware reset entry point to the application entry function.

Code Layout and 8 KiB Macro Header Motivation

The application binary within each slot, in case of TC375TP96F300, is organized from the application start address (`slot_base` + `0x2000`) as follows:

1. INTVEC0 — 8 KiB, 8 KiB-aligned.
2. INTVEC1 — 8 KiB, 8 KiB-aligned.
3. INTVEC2 — 8 KiB, 8 KiB-aligned.
4. TRAPVEC0 — 256 bytes, 256-byte-aligned.
5. TRAPVEC1 — 256 bytes, 256-byte-aligned.
6. TRAPVEC2 — 256 bytes, 256-byte-aligned.
7. START_CORE0 — 32 bytes (jump sequence to application entry).

8. START_CORE1 — 32 bytes.
9. START_CORE2 — 32 bytes.
10. `.text` / `.rodata` — remaining application code and read-only data.

Figure 3.3 shows the memory layout for Slot A (base address `0xA0300000`). The application binary initiates at `0xA0302000`. Because this starting address aligns to an 8 KiB boundary and the IVT structure spans exactly 8 KiB, the first IVT occupies the `0xA0302000`–`0xA0303FFF` range without requiring internal or external padding. Subsequent IVTs map to contiguous 8 KiB intervals. The three TVTs originate at `0xA0308000`, spaced at 256-byte intervals, while the three startup vectors allocate at `0xA0308300` in 32-byte intervals.

The metadata header (Section 3.3.5) maps the `core_pc` entries directly to the respective startup vector addresses: `0xA0308300` (CPU0), `0xA0308320` (CPU1), and `0xA0308340` (CPU2). These startup vectors execute the hardware jump instructions that transfer execution control to the application entry function.

The 8 KiB macro header satisfies two constraints simultaneously: it accommodates the public key and digital signature for all supported cryptographic algorithms, and it positions the application binary at an 8 KiB-aligned base address, meeting the IVT alignment requirement without additional padding.

3.3.5 Application Metadata Header

Every firmware image is prefixed by a 96-byte metadata header that carries all parameters required for boot election, signature verification and startup procedure. Table 3.3 lists the complete field layout. All multi-byte integer fields use little-endian byte order.

The `core_pc` array contains 32-bit entry-point addresses for up to six TriCore CPUs. The TC375TP exposes three active cores (CPU0, CPU1, CPU2); the remaining three entries are set to `0xFFFFFFFF`, the erased-flash pattern, which is not a valid entry-point address. The bootloader reads core entry points for the application that will be executed directly from this array.

The metadata header occupies two positions within the 8 KiB macro header block. An identical copy of the 96-byte header is placed at offset zero (the *primary* header, parsed first) and at offset $8192 - 96 = 8096$ bytes from the start of the macro header block (the *copy* header, adjacent to the application binary). The digital signature covers the concatenation of the bottom header and the full application binary, so any modification to either is detected by verification. The byte-for-byte equality check between primary and copied headers performed before signature verification ensures that an attacker cannot substitute a valid primary header to pass early structural checks while presenting a different bottom header to the signature verifier.

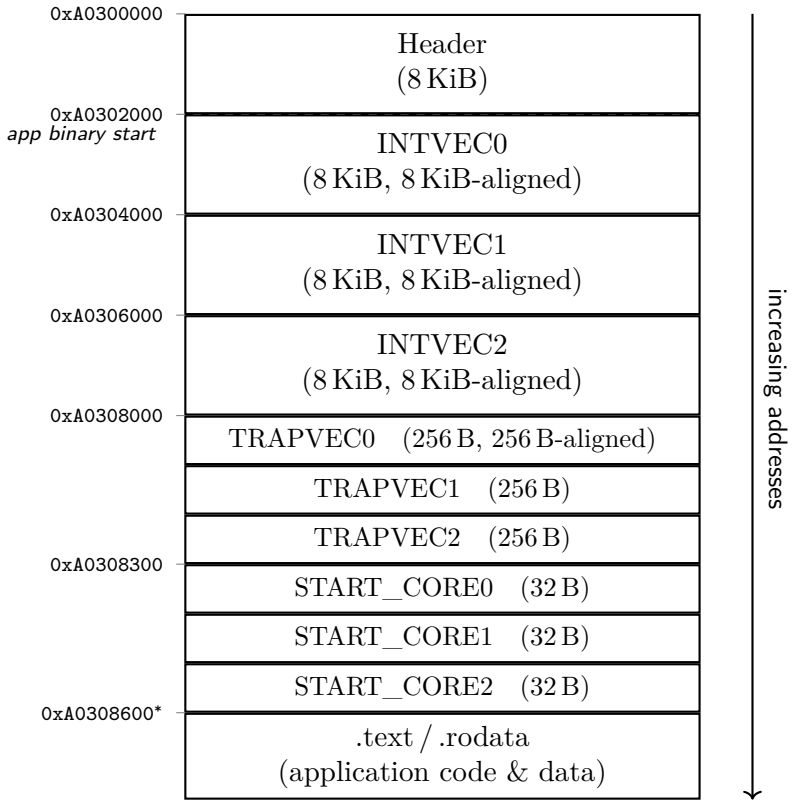


Figure 3.3: Memory layout of an AURIX TC375TP firmware slot (Slot A, base address `0xA0300000`). The 8 KiB metadata header precedes the application binary, which begins at `0xA0302000` where the first INTVEC lands on a natural 8 KiB-aligned boundary. *Approximate start of `.text`.

Table 3.3: Metadata header field layout (96 bytes, little-endian).

Offset (bytes)	Size (bytes)	Field	Description
0	4	`magic`	ASCII identifier `AURX`
4	4	`target_platform`	Target Platform CHID
8	4	`binapp_size`	Application binary size in bytes
12	1	`header_size`	Metadata header size in bytes (96)
13	1	*(padding)*	Reserved
14	2	`build_id`	Build identifier
16	1	`version_major`	Semantic version: major
17	1	`version_minor`	Semantic version: minor
18	1	`version_patch`	Semantic version: patch
19	1	`algo_id`	Signature algorithm identifier (Table 3.6)
20	1	`hash_id`	Hash algorithm identifier
21	1	*(padding)*	Reserved
22	2	`pubkey_size`	Public key size in bytes
24	2	`signature_size`	Signature size in bytes
26	6	*(padding)*	Reserved
32	24	`core_pc[6]`	Program counter address for each of six TriCore cores
56	8	*(padding)*	Reserved
64	32	`binapp_hash`	SHA3-256 hash of the application binary; not used by the current verification path, reserved for future use

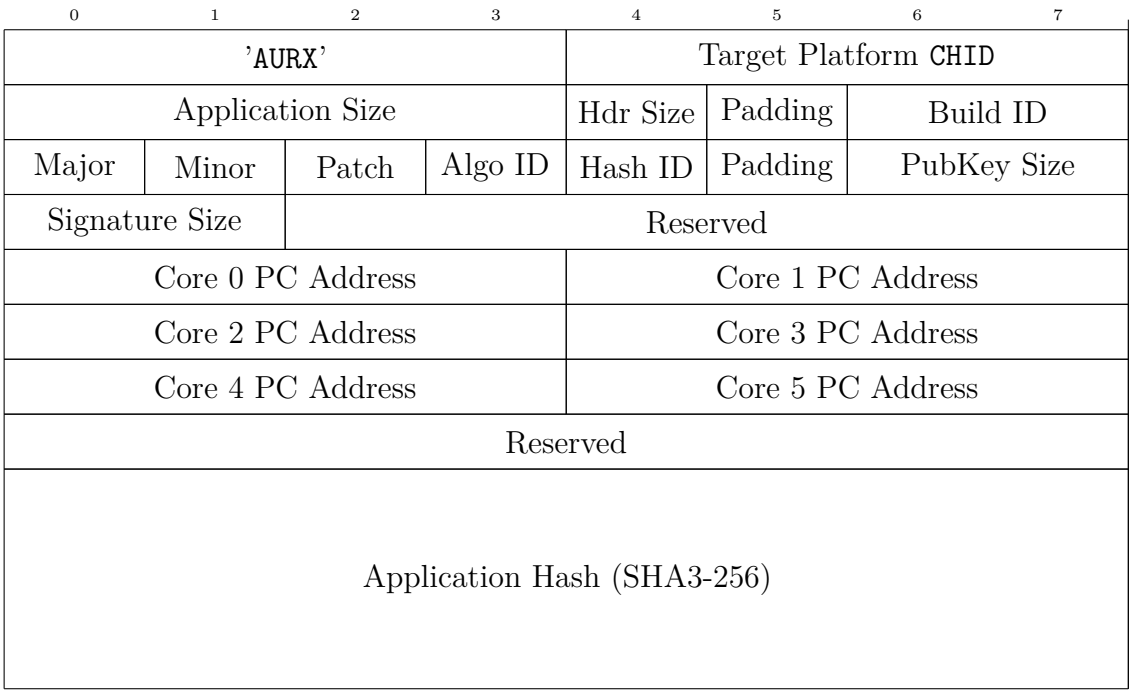


Figure 3.4: Bootloader Metadata Header Layout (8 Bytes per Row)

Macro Header Memory Layout

Within the 8 KiB macro header block, the four regions are arranged as shown in Table 3.4 and Figure 3.5.

Table 3.4: Macro Header Memory Layout (8192 Bytes Total).

Offset (Bytes)	Size (Bytes)	Field	Description
0	96	`primary_header`	Bootloader metadata header.
96	Variable	`public_key`	Public key with optional 32-byte alignment padding.
O_1	Variable	`signature`	Cryptographic signature with optional 32-byte alignment padding.
O_2	Variable	`padding`	Zero padding extending to byte offset 8096.
8096	96	`copied_header`	Copy of the primary metadata header.

The public key begins at offset 96, immediately after the primary metadata header. The offset O_1 of the signature is computed by rounding the end of the public key region up to the next 32-byte boundary. The offset O_2 of the zero-padding region is computed by rounding the end of the signature region up to the next 32-byte boundary. The zero-padding region extends from O_2 to offset 8096, where the bottom metadata header begins. The sizes of the public key and signature fields are not constants; they are read from the `pubkey_size` and `signature_size` fields of the primary metadata header, which are in turn determined by the `algo_id` field (Table 3.6).

3.3.6 Boot Decision Algorithm

This section describes how the bootloader selects the firmware image to execute. The decision proceeds from a flash-presence check through structural validation and version ordering to the final candidate selection and the copy-on-update policy.

Flash Presence Check

Before extracting the metadata header, the `parse_slot()` function verifies that the memory region containing the macro header is programmed and not erased. It

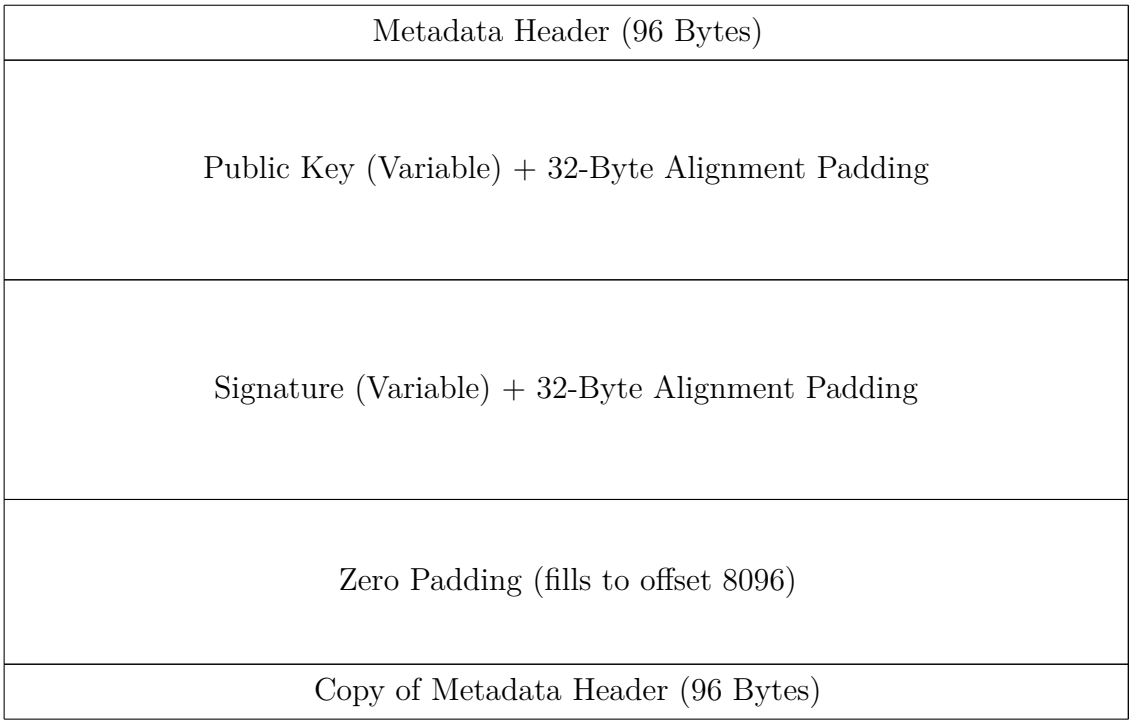


Figure 3.5: Macro Header High-Level Memory Map.

invokes the `is_pflash_range_written()` function, which issues hardware erase-verify commands across the macro header block. If the macro header region returns an erased state, the bootloader classifies the entire firmware slot as unprogrammed and bypasses it. This pre-validation prevents the generation of hardware memory access traps that occur when reading from unprogrammed flash sectors.

Structural Validation

Following a successful flash presence check, the `parse_slot()` function executes the structural validation for the candidate slot. Upon successful validation, the function populates a `slot_data_t` descriptor. The validation sequence proceeds in the following order:

Magic bytes. The four-byte field at offset zero is compared against the expected sequence `AURX`.

Algorithm and hash identifiers. Both `algo_id` and `hash_id` must fall within $[1, \texttt{ALGO_MAX} - 1]$ (Table 3.6); out-of-range values cause the slot to be rejected.

Size constraints. The declared `pubkey_size` and `signature_size` (Section 3.3.5) must fit the 8 KiB macro header: the sum of the top header (96 bytes), the 32-byte-aligned public key region, the 32-byte-aligned signature region, and the bottom header (96 bytes) must not exceed 8192 bytes. The declared application binary size must not exceed $1{,}572{,}864 - 8{,}192 = 1{,}564{,}672$ bytes, the usable binary region within a 1.5 MiB slot.

Top/bottom header consistency. The parser compares the 96-byte primary header byte-for-byte against the copy at offset 8096; a mismatch causes validation to fail with `PARSE_ERR_HEADER_MISMATCH`.

On success, `parse_slot()` resolves the in-flash addresses of the public key, signature, bottom header, and application binary. The signature pointer is placed at the first 32-byte-aligned offset following the end of the public key region.

Fault Attack Mitigation: Sentinel Constants

The status and candidate values used in the boot decision logic are 16-bit constants chosen so that a bit fault in a stored status does not turn a rejection into an acceptance [11]. The accept value `SLOT_VALID` (which is equivalent to `PARSE_OK`, `0x3A5C`) and the base reject value `SLOT_INVALID` (`0xC5A3`) are bitwise complements with a Hamming distance of 16:

```
#define SLOT_VALID     0x3A5C
#define SLOT_INVALID   0xC5A3
#define CANDIDATE_NONE 0xC5A3
#define CANDIDATE_A    0xAAAA
#define CANDIDATE_B    0xBBBB
```

The error codes `PARSE_ERR_*` span from `0xC5A3` to `0xC5AB` and are numbered sequentially, but they all share the high byte `0xC5`, which is the complement of the accept high byte `0x3A`. Every reject value therefore differs from the accept value in at least its eight most significant bits, requiring a fault to flip at least eight specific bits simultaneously to turn a rejection into the accept value.

The boot logic evaluates the decision as an equality test against the single accept value (`status == PARSE_OK`) and treats all other values as rejections, so a corrupted status defaults to a rejection. The candidate identifiers `CANDIDATE_A` (`0xAAAA`) and `CANDIDATE_B` (`0xBBBB`) use alternating bit patterns distinct from the status values. Because they select between two slots that have already passed signature verification, a fault that swaps them changes only which valid image boots and does not affect the authentication decision. This implementation follows the software fault-mitigation guidance from Infineon for safety-critical TriCore applications operating under electromagnetic interference and power glitching conditions [11].

Candidate Selection and Version Ordering

After validation of both slots, the bootloader selects a firmware candidate using the algorithm in Algorithm 10, whose selection logic is also given in decision table form in Table 3.5. A slot qualifies as viable if its structural validation succeeded and its version is greater than or equal to the minimum version threshold (*current_major*, *current_minor*, *current_patch*). Version comparison uses strict lexicographic ordering on the triple (major, minor, patch).

The minimum version threshold is the version of the most recent firmware the device has accepted. It is kept in the first sector of DFLASH0, the rewritable data flash, and the bootloader reads it at startup. When a newer firmware passes verification and is selected for boot, the bootloader overwrites the stored version with the newer one in this dedicated DFLASH0 region; a firmware whose version equals the stored one leaves it unchanged, and an older one is refused. Monotonicity is enforced in software by this read-compare-write sequence rather than by hardware. In a production deployment with HSM access, the threshold would instead reside in DFLASH1, the data flash reserved for exclusive HSM access, isolating it from the application cores together with the trust anchor. Without an HSM, the same isolation is obtained by running the bootloader in a privileged mode and denying the application cores write access to the counter sector through the Memory Protection Unit [51].

Signature verification is invoked exclusively for slots that pass structural validation and the version threshold check. When both slots are viable, the algorithm verifies the higher-version slot first and falls back to the lower-version slot only if the primary candidate carries an invalid signature. This ordering minimizes the

Algorithm 10 Boot Candidate Selection

Input: `status_A`, `status_B`, `slot_data[2]`, minimum version $v_{\min}$
Output: `candidate` $\in$ {CandidateA, CandidateB, None}

$a_ok \leftarrow (\texttt{status_A} = \text{SlotValid}) \wedge (\text{ver}(A) \geq v_{\min})$
$b_ok \leftarrow (\texttt{status_B} = \text{SlotValid}) \wedge (\text{ver}(B) \geq v_{\min})$
if $a_ok \wedge b_ok$ **then**
    if $\text{ver}(B) > \text{ver}(A)$ **then**
        $r \leftarrow \text{VerifySignature}(B)$
        if $r = \text{OK}$ **then return** CandidateB
        end if
        $r \leftarrow \text{VerifySignature}(A)$
        if $r = \text{OK}$ **then return** CandidateA
        end if
    else
        $r \leftarrow \text{VerifySignature}(A)$
        if $r = \text{OK}$ **then return** CandidateA
        end if
        $r \leftarrow \text{VerifySignature}(B)$
        if $r = \text{OK}$ **then return** CandidateB
        end if
    end if
else if a_ok **then**
    $r \leftarrow \text{VerifySignature}(A)$
    if $r = \text{OK}$ **then return** CandidateA
    end if
else if b_ok **then**
    $r \leftarrow \text{VerifySignature}(B)$
    if $r = \text{OK}$ **then return** CandidateB
    end if
end if
return None

Table 3.5: Boot Candidate Selection Decision Table

	Both ok, v(B) > v(A)			Both ok, v(A) ≥ v(B)			Only A		Only B		None
	R1	*R2*	*R3*	*R4*	*R5*	*R6*	*R7*	*R8*	*R9*	*R10*	*R11*
Conditions											
`a_ok`	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N
`b_ok`	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N
v(B) > v(A)	Y	Y	Y	N	N	N	–	–	–	–	–
sig(B)OK	Y	N	N	–	Y	N	–	–	Y	N	–
sig(A)OK	–	Y	N	Y	N	N	Y	N	–	–	–
Actions											
Choose A		✓		✓			✓				
Choose B	✓				✓				✓		
NONE			✓			✓		✓		✓	✓

– = don't care (condition not evaluated); ✓ = outcome for this rule.

number of signature verifications in the common case: a single verification suffices when the preferred slot is valid.

Copy-on-Update Policy

When the selected candidate is Slot B, the bootloader executes a full copy of Slot B into Slot A before any CPU is released to the application.

The `wipe_and_copy_slotB_into_slotA()` function first erases the minimum number of 16 KiB Slot A sectors required to accommodate the total transfer size $8192 + $ `binapp_size` bytes, then writes 32-byte pages sequentially from Slot B into Slot A. After the copy completes, Slot A is re-parsed and its signature is verified again on the copied contents, so that the bytes about to execute are confirmed to be the same authenticated bytes previously verified in Slot B. If the re-parse or the signature re-verification fails, the candidate is demoted to `CANDIDATE_NONE` and the update subsystem is invoked. This second verification closes the time-of-check to time-of-use window between the verification of Slot B and execution from Slot A, in which an incomplete or errored copy could otherwise leave unverified bytes in the execution slot.

This path is taken only when Slot B is the selected candidate, which occurs on the first boot after a firmware update. Routine boots find a valid Slot A and proceed directly to execution without any copy. Once the updated image is copied and re-verified, every subsequent boot takes this Slot A path until the next update. The copy therefore runs at most once per firmware update, not on every boot, and lies outside the boot latency reported in Chapter 4.

3.3.7 Cryptographic Verification

This section describes the cryptographic authentication of a firmware candidate: the root-of-trust mechanism, the supported signature algorithms, and the verification interface invoked by the boot decision.

Root of Trust Mechanism

The root of trust consists of one SHA3-256 hash of the trusted public key per supported algorithm type, stored as read-only constants in the `.rodata` section of the bootloader binary. The trust anchor table `pk_sha3_256_v` is indexed by `algo_id`, so each algorithm variant maintains an independent public key hash. An entry of `NULL` for a given algorithm identifier causes verification to fail immediately without performing any comparison. The classical algorithms (Ed25519 and RSA-PSS) are included only as performance baselines for this evaluation; a deployed bootloader would omit their entries entirely, leaving only the post-quantum algorithms in the

table. The constants are generated by the `keygen` subcommand of the signing tool and compiled into the bootloader at build time.

At runtime, `verify_signature_of_parsed_slot()` computes the SHA3-256 hash of the candidate slot public key and compares it against the stored reference via constant-time comparison. A mismatch rejects the slot with `PARSE_ERR_ROT_FAILED` before signature verification is invoked; the early rejection precludes key pair substitution.

Supported Algorithms

Table 3.6 lists the ten signature algorithms supported by the bootloader. The ML-DSA family (parameter sets 44, 65, and 87), FN-DSA-512, and FN-DSA-1024 use implementations from the PQClean library [52]. HAWK-512 and HAWK-1024 use the HAWK team reference C implementation, in the 2025 revision aligned with the version 1.1 specification [35]. The Ed25519 implementation is derived from the SUPERCOP `ref10` reference implementation [53]. RSA-PSS-2048 and RSA-PSS-3072 use the BearSSL library [54], which provides a stack-only RSA implementation optimized for embedded targets.

Table 3.6: Supported signature algorithms. PQC indicates a post-quantum scheme. Key and signature sizes are in bytes.

ID	Identifier	Algorithm	PQC	Public key	Signature
1	`ALGO_MLDSA44`	ML-DSA-44 [31]	Yes	1312	2420
2	`ALGO_ED25519`	Ed25519	No	32	64
3	`ALGO_FALCON512`	FN-DSA-512 [33]	Yes	897	666
4	`ALGO_FALCON1024`	FN-DSA-1024 [33]	Yes	1793	1280
5	`ALGO_HAWK512`	HAWK-512	Yes	1024	555
6	`ALGO_HAWK1024`	HAWK-1024	Yes	2440	1221
7	`ALGO_RSA_PSS2048`	RSA-PSS-2048 [54]	No	256	256
8	`ALGO_RSA_PSS3072`	RSA-PSS-3072 [54]	No	384	384
9	`ALGO_MLDSA65`	ML-DSA-65 [31]	Yes	1952	3309
10	`ALGO_MLDSA87`	ML-DSA-87 [31]	Yes	2592	4627

RSA-PSS is benchmarked using two public exponents for each key size; the exponent is carried in the public key. ML-DSA is additionally measured in a portable C build and an assembly-optimized build, which yields fifteen distinct timing entries in Chapter 4.

Cryptographic API

The bootloader uses the PQClean extended API [52], which is a superset of the original NIST PQC standardization API. The original NIST API defines three functions:

- `crypto_sign_keypair(pk, sk)`: generates a public/private key pair.
- `crypto_sign(sm, smlen, m, mlen, sk)`: produces a signed message bundle in which the signature is prepended to the message.
- `crypto_sign_open(m, mlen, sm, smlen, pk)`: verifies the signed message bundle and extracts the original message.

The functions in the original API set are inconvinient for this bootloader. These functions bundle together the signature and the message in the same byte array. Because the signature resides in the 8 KiB header and the firmware binary occupies the application slot, these are incompatible.

The PQClean extended API adds two functions that decouple the signature from the message:

- `crypto_sign_signature(sig, siglen, m, mlen, sk)`: generates a detached signature over the message; the message is not appended to the output.
- `crypto_sign_verify(sig, siglen, m, mlen, pk)`: verifies a detached signature over the message without extracting the message from the signature object (none is present).

The signing tool `sign.py` uses `crypto_sign_signature` via the `pyqcsign` extension module. During verification, the bootloader dynamically executes the appropriate `crypto_sign_verify` function via a function-pointer dispatch table.

Not all algorithm implementations provide the extended API natively. For HAWK, Ed25519, and RSA-PSS, the functions `crypto_sign_signature` and `crypto_sign_verify` were added manually, following the PQClean calling convention.

Signature Verification Call

The ten supported verification functions share the same prototype of function type

```
1 int (*verify_fn_t)(const uint8_t *sig, size_t siglen,
2                    const uint8_t *m, size_t mlen,
3                    const uint8_t *pk)
```

indexed by algorithm identifier in a static dispatch table, an array of function pointers. The verification target message is the concatenation of the bottom metadata header and the application binary, both located contiguously in flash. The message length equals `header_size` + `binapp_size` bytes. Each algorithm-specific wrapper casts the `size_t` arguments to the type expected by its underlying library to resolve mismatches between libraries sourced from PQClean (`size_t`), HAWK (`unsigned long long`), and the Ed25519 and RSA-PSS reference implementations.

A failed signature verification causes the slot to be rejected with `PARSE_ERR_BAD_SIGNATURE`. The field `sig_validity` in the `slot_data_t` descriptor is set to 1 on success and 0 on failure; its initial value of -1 marks slots not yet subjected to cryptographic verification.

3.3.8 Optimized NTT Implementation

The ML-DSA signature scheme requires repeated polynomial multiplications in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^{256}+1)$ with $q = 8{,}380{,}417$. The algorithm evaluates the polynomial at the odd powers of the primitive 512-th root of unity $\zeta = 1{,}753$. This Number Theoretic Transform (NTT) implementation for the AURIX TC3xx adapts execution strategies developed for the ARM Cortex-M4 in the `pqm4` library [55].

Architectural Limitations and Register Allocation The `pqm4` Cortex-M4 implementation caches zeta values in the FPU registers (`S2`–`S8`), exploiting their availability when no floating-point arithmetic is required. The TriCore FPU shares the standard data registers D0–D15 for floating-point operations; there is no separate FPU register file (unlike the ARM Cortex-M4's `S0`–`S31`) that can be used for caching. The port therefore maps these seven zeta values to address registers `A2`, `A3`, `A6`, `A7`, `A11`, `A12`, and `A13`. All registers used by the routine, including the return-address register `A11`, are saved on entry and restored before return, so the reuse is transparent to the caller.

Unlike the Cortex-M4, which uses the `VLDM` instruction for bulk data loads, the TriCore requires seven sequential `LD.A` instructions to load the cache. Furthermore, TriCore arithmetic instructions (`MUL`, `MADD`) operate strictly on data registers (`D0`–`D15`). Before executing a multiplication, the algorithm must transfer the zetas from the address register to a data register using a `MOV.D` instruction. Table 3.7 details this resource allocation.

To reduce memory access latency, the algorithm applies layer merging [56, 55]. This technique executes multiple NTT layers within a single loop iteration, storing intermediate values in processor registers. For an 8-layer NTT processing 256 coefficients, the algorithm uses a 3-3-2 layer merging structure:

- **Phases 1–3 (3 layers):** Loads eight coefficients into data registers. Executes

Table 3.7: Comparative architectural resource utilization for the 3-3-2 merged NTT.

Resource	**ARM Cortex-M4**	**Infineon TriCore TC3xx**
Integer/data registers for polynomial	8 (`R5`–`R8`, `R11`, `R12`, `R14`, `R4`)	8 (`D8`–`D15`)
Integer/data registers for arithmetic scratch	2 (`R9`, `R10`)	4 (`D0`–`D3` in Montgomery, `D6`–`D7` scratch)
Constants and control registers	2 (`R2` q^{-1}, `R3` q)	2 (`D4` q, `D5` q^{-1})
Pointer registers	2 (`R0`, `R1`)	2 (`A4`, `A5`)
Zetas cache (3-layer)	7 FPU registers (`S2`–`S8`)	7 Address registers (`A2`, `A3`, `A6`, `A7`, `A11`, `A12`, `A13`)
Loop limit storage	2 FPU registers (`S9`, `S10`) + 1 pointer (`S0`)	2 Address registers (`A14`, `A15`) (inherent)
Instructions for zetas load	1 (`VLDM`, bulk)	7 (Sequential `LD.A`)
Instructions for Montgomery multiplication	3 (`SMULL`, `MUL`, `SMLAL`)	4 (`MUL E0`, `MUL`, `MADD E0`, `MOV`)

three butterfly layers on pairs separated by distances of 128, 64, and 32, then stores the results.

- **Phases 4–6 (3 layers):** Processes independent eight-coefficient blocks with butterfly operations at distances of 16, 8, and 4.
- **Phases 7–8 (2 layers):** Evaluates four-coefficient blocks with final butterfly operations at distances of 2 and 1.

Performance Evaluation

Table 3.8 reports execution cycle counts for the forward NTT and Inverse NTT (INTT) on the TC375TP microcontroller. Measurements compare the FIPS 204 C reference against the optimized assembly port. To guarantee deterministic memory access, the test disabled the program cache and allocated all arrays within the Data Scratch-Pad RAM (DSPR).

Table 3.8: ML-DSA NTT and INTT cycle counts on the TC375TP microcontroller.

Operation	C Reference (cycles)	ASM (cycles)	Reduction
Forward NTT	260,570	15,339	94.1%
Inverse NTT (INTT)	300,070	17,701	94.1%

The assembly implementation reduces both forward NTT and INTT execution times by 94.1% (from 260,570 to 15,339 cycles, and 300,070 to 17,701 cycles, respectively). The 3-3-2 loop structure maximizes data register reuse. Additionally, writing the algorithm in assembly eliminates the stack-based prologue and epilogue overhead required by C compiler conventions.

3.3.9 Multi-Core Application Startup

After a valid candidate in Slot A is confirmed, the bootloader configures and releases the secondary cores before transferring its own execution to the application entry point.

The startup sequence proceeds as follows.

First, `IfxCpu_setAllIdleExceptMasterCpu()` places CPU1 and CPU2 into the idle state, ensuring both cores are halted and their pipeline state is clean before any program counter assignment.

Second, a loop over CPU indices 1 through `IFXCPU_NUM_MODULES` − 1 calls `IfxCpu_setProgramCounter()` for each secondary core, loading the entry point address read from the `core_pc` array in the candidate slot descriptor.

Third, a second loop calls `IfxCpu_startCore()` followed by `IfxCpu_setCoreMode(run)` for each secondary core, clearing the BHALT bit and releasing the cores to begin execution at their assigned entry points.

Fourth, `Ifx_Ssw_jumpToFunction()` transfers CPU0 to the application entry point at `core_pc[0]`. This call does not return; any instruction following it is unreachable.

Secondary cores are released before CPU0 jumps so that all three cores begin executing the application within a bounded window. The exact synchronization between cores after the jump is the responsibility of the application startup software, which is outside the scope of the bootloader.

3.3.10 Update Subsystem

The update subsystem receives firmware images from a user-initiated request over the Ethernet interface. The firmware header contains all required metadata in the 8 KiB header preceding the binary.

The update subsystem is activated when the dedicated GPIO button is pressed during the boot initialization phase in `core0_main()`: Slot B is erased and `startUpdateSubsystem()` is called unconditionally, bypassing the boot decision algorithm. This trigger provides recovery when both slots fail verification.

A software trigger activates when the boot decision algorithm produces `CANDIDATE_NONE`, indicating that no slot passed both structural validation and signature verification. The bootloader then enters a loop that alternates LED signaling with calls to `startUpdateSubsystem()`, until the host delivers a new firmware image and a software reset reinitiates the boot sequence.

Network Firmware Upload

The update subsystem initializes the Ethernet interface and the lwIP TCP/IP stack, then starts a listening TCP server via `netIfInit_and_listen()` with a static IP address (`10.10.10.10`) on port `80`. A System Timer Module 0 (STM0) comparator generates a 1 ms interrupt that drives the lwIP timer and polling loops while the upload is in progress.

The application header (8 KiB) and application binary are received as a single contiguous stream and written to Slot B flash starting at `0xA0480000`. The network handler sets `upload_finished` variable on receipt of the complete application image.

After `upload_finished` is set, the update subsystem polls the lwIP stack for 500 ms to flush any remaining TCP send buffers before reset. The STM0 interrupt is then disabled and `trigger_sw_reset()` issues a system software reset; the

subsequent boot cycle re-runs the boot decision algorithm against the freshly written Slot B image.

3.3.11 Signing Tool and Trust Anchor Provisioning

This section describes the host-side tooling that prepares a signed firmware image and provisions the trust anchor compiled into the bootloader.

Firmware Signing Workflow

The host-side utility `sign.py` produces the final firmware image ready for the upload.

The signing workflow proceeds in nine steps:

1. Load the Intel HEX file produced by the TASKING VX-toolset compiler with the aid of `intelhex` library.
2. Merge cached (`0x80302000`) and non-cached (`0xA0302000`) address regions into a unified binary view, verifying consistency between any data appearing at both aliases (see subsection 3.3.11 below).
3. Extract the program flash bytes from the unified view.
4. Pad the binary to a 32-byte boundary using `0xFF` bytes.
5. Compute the SHA3-256 hash of the padded binary.
6. Construct the 96-byte metadata header, embedding the hash, the declared algorithm identifier, the public key and signature sizes, and the core entry-point addresses.
7. Compute the digital signature over the concatenation of the metadata header and the padded application binary through `pyqcsign`, a CPython extension module written against the Python C API, which wraps the key generation, signature and verification functions of every supported scheme. The module exposes the PQClean extended function `crypto_sign_signature` for all supported algorithms.
8. Immediately verify the signature using `crypto_sign_verify` to confirm the signing operation produced a consistent result.
9. Assemble the 8 KiB macro header: primary metadata (96 bytes) | public key (32-byte aligned) | signature (32-byte aligned) | zero padding | bottom metadata copy (96 bytes at offset $8192 - 96$).

An `ALGO_IDS` dictionary maps algorithm names to the numeric identifiers in Table 3.6.

Intel HEX Cached/Non-Cached Address Aliasing

The TASKING toolchain generates ELF and Intel HEX files that describe the program code at the cached address alias (starting at `0x80302000`) and at the non-cached alias (starting at `0xA0302000`). Because the two aliases refer to the same physical flash, the signing tool presents the firmware as a single contiguous binary to the internal hash and signature generation functions.

The `merge_hex_regions()` function remaps any data at the cached alias into the non-cached address space and verifies that bytes appearing at both aliases are identical. Inconsistent data at the two aliases indicates a toolchain error and causes the signing operation to abort. The merged non-cached view serves as the main binary representation for all subsequent operations.

Root of Trust Provisioning

The `keygen` subcommand generates a cryptographic key pair for the selected algorithm and writes two output files: the key pair in the signing tool's native format and a C source file `<algo>_rot.c` containing a `const uint8_t` array of the SHA3-256 hash of the public key. This file is compiled directly into the bootloader binary, establishing the trust anchor in the `.rodata` section at build time. The trust anchor array is indexed by algorithm identifier to allow the bootloader to select the correct reference hash at runtime based on the `algo_id` field of the candidate slot header.

Embedding the trust anchor within the mutable bootloader firmware image represents a development-phase compromise. In production environments, it is recommended to store the trust anchor in read-only memory, and the bootloader should reference this protected region.

Chapter 4

Benchmark

This chapter reports the measured performance of the evaluated schemes on the target microcontroller. The first sections describe the measurement setup and the hash baseline. The following sections report the results for key generation, signing, and verification; each one carries a table that combines timing, size of the cryptographic material, and stack depth, plus the ROM of the verify path for verification. The last sections quantify the effect of the assembly NTT, report the ROM footprint of the full libraries, and discuss the findings, including the comparison at matched security levels.

4.1 Benchmark Setup

All benchmarks execute on the Infineon AURIX TC375TP microcontroller. The active core is CPU0, a TriCore 1.6.2P running at 300 MHz. The program cache is disabled on all test runs, forcing instruction fetches directly from program flash in deterministic time. All temporary memory allocations are in the CPU0 Data Scratch-Pad RAM (DSPR).

4.1.1 Measurement Methodology

Clock cycle counts are obtained from the TriCore internal performance counter register `CCNT`. The counter is a 32-bit register that uses 31 bits for the cycle count and one sticky overflow bit. When the counted value reaches $2^{31} - 1 = 2{,}147{,}483{,}647$ cycles, the overflow bit is set and the measurement is invalid. At 300 MHz, this threshold corresponds to approximately 7.16 seconds. In the tables below, the label *OVF* marks measurements that reached this limit; for these runs only the lower bound of 7.16 s is known. Overflowed iterations are excluded from the mean and standard deviation, and percentile statistics and are reported as a count: all RSA

key generation instances and 284 of 1000 instances of Falcon-1024 key generation overflowed. For Falcon-1024 key generation, the only operation where some but not all runs overflow, the median over all 1000 runs is reported alongside the overflow count. The overflowed runs are counted as larger than the counter limit, which keeps the median a measured value because fewer than half of the runs overflowed.

Each algorithm executes 1000 iterations. Sign and verify timings are split into two independently measured phases: hash absorbtion phase and cryptographic operation. The first phase (`sign_hash`, `verif_hash`) covers the message hash step. The second phase (`sign_sign`, `verif_verif`) covers the cryptographic signature computation or verification. This separation isolates message-length-dependent cost from algorithm-dependent cost. Key generation is measured as a single phase.

4.1.2 Test Message

The signed message is a 65,536-byte (64 KiB) buffer of pseudo-random data written to a non-cached alias of the TC375TP Program Flash 1 bank (`0xA0300000`). The latency of the cryptographic operations only is indipendent from the size of the firmware. The execution time of the hash absorb phase scales linearly with the size of the message, in this case the firmware. For example, in the absorb phase, a 1 MiB firmware image requires approximately 16× the time for the 64 KiB one.

The hash phase is measured separately for every scheme: SHA-256 for RSA-PSS is timed in isolation, then BearSSL operates on the resulting 32-byte digest. RSA hash figures are therefore directly comparable to those of the other algorithms.

4.1.3 Hash Libraries

Different algorithms use different hash implementations: ML-DSA and Falcon use the Falcon project's SHAKE-256 implementation, which applies loop unrolling and is faster than the PQClean-native version. HAWK uses its own Keccak implementation, optimized for HAWK's internal permutation schedule. Ed25519 uses the SHA-512 implementation from SUPERCOP ref10 [53] (FIPS 180-4). RSA-PSS uses the SHA-256 implementation provided with BearSSL [54].

A timing difference of $\sim$ 1.8 ms in the hash phase is present between the ML-DSA reference and assembly builds despite both using the same SHAKE-256 implementation. This delta is consistent between all three ML-DSA parameter sets and both sign and verify hash measurements. This offset is attributable to binary code layout. The builds place the Keccak inner loop at different addresses in uncached program flash, altering the instruction fetch alignment.

4.1.4 Algorithm Configurations

The benchmarked algorithms are ML-DSA-44/65/87 (PQClean, C reference and asm-NTT variant), Falcon-512/1024 (PQClean), HAWK-512/1024 (HAWK v1.1 reference implementation), Ed25519 (SUPERCOP ref10), RSA-PSS-2048 and RSA-PSS-3072 (BearSSL, $e = 3$ and $e = 65537$). These are reference implementations rather than platform-optimized code, with the exception of the ML-DSA assembly NTT. The reported timings are therefore a conservative upper bound: an implementation tuned for the target would only reduce them. ML-DSA is the only algorithm benchmarked in two builds: the PQClean C reference and an assembly-optimized version that replaces the forward and inverse NTT with a TriCore assembly port (see Section 4.6 and [55]). All algorithms run 1000 iterations, except for RSA. RSA-PSS-2048 runs 10 iterations; RSA-PSS-3072 runs 3 iterations. RSA timing is constant to within 300 cycles across all iterations.

4.1.5 Memory Metrics

The TC375TP CPU0 SRAM is 240 KiB. The TASKING linker script for the bootloader allocates this as: 8 KiB for the Context Save Area (CSA), 125 KiB for the user stack (`USTACK0`), 1 KiB for the interrupt stack (`ISTACK0`), 4 KiB for the heap, and the remaining 102 KiB for static data. The user stack budget for all cryptographic operations is therefore 125 KiB.

The stack columns of Tables 4.2, 4.3 and 4.4 report the worst-case stack depth of each operation, extracted from the linker call graph. The value reported is the total stack used by the function and all functions it calls (the B field in the TASKING call graph output). The `randombytes` DRBG function is excluded from these figures (it is flagged as reported elsewhere in the call graph to avoid double-counting); its stack contribution is estimated below 1 KiB and does not affect the budget analysis.

The ROM figures count the cryptographic library code only, excluding the benchmark harness and DRBG. The figures are taken from the section of the linker map, filtered to include only the algorithm's own object files. The complete bootloader image, with the lwIP stack, the iLLD runtime, the hash code, the DRBG, and the startup code, occupies approximately 250 KiB (Section 3.3.2).

4.2 Hash Baseline

Table 4.1 reports the mean hash time over the 65,536-byte test message by hash function. The hash timing is deterministic for a fixed message: across 1000 iterations, the value varies by at most tens of cycles.

Table 4.1: Mean hash time for the 65,536-byte test message by hash function.

Hash	**Used by**	**Mean (cycles)**	**Mean (ms)**
SHAKE-256 (Falcon)	ML-DSA(ref),Falcon	∼16,380,000	∼54.6
SHAKE-256 (Falcon)	ML-DSA(asm)	∼15,830,000	∼52.8
Keccak (HAWK impl.)	HAWK	∼16,460,000	∼54.9
SHA-512 (SUPERCOP)	Ed25519	8,599,889	28.7
SHA-256 (BearSSL)	RSA-PSS	10,787,971	36.0

The SHAKE-256 and Keccak figures are representative means; the message-hash cost varies slightly by scheme. Per-variant values appear in Tables A.2 and A.4.

SHA-512 processes the 64 KiB message in 512 blocks of 128 bytes, at approximately 16,797 cycles per block. SHAKE-256 with a 136-byte rate processes the same message in 482 permutations, at approximately 33,900 cycles per permutation. SHA-512 is therefore approximately 1.9× faster than SHAKE-256 for this message length on the TC375TP. This difference reflects the relative efficiency of SHA-512 versus Keccak on 32-bit integer hardware without hardware acceleration. SHA-256 processes the same 64 KiB message in 1,024 blocks of 64 bytes, at approximately 10,535 cycles per block (36.0 ms total).

4.3 Key Generation

Key generation runs once on the developer machine during key provisioning; the bootloader never executes it at boot time. Table 4.2 reports the mean and standard deviation of key generation time for every benchmarked variant, together with the mean and maximum number of solver attempts for Falcon and HAWK, the size of the generated key pair, and the stack depth of the call.

The iteration count (Tries) is not applicable for ML-DSA and Ed25519. While ML-DSA utilizes rejection sampling internally to generate valid polynomial coefficients, the overall key generation procedure does not require restarting. Similarly, Ed25519 key generation is successful for any given seed.

There are two type of keypair generation in these lattice schemes. The ML-DSA key generation algorithm constructs cryptographic keys via coefficient-level rejection sampling: the standard deviation stays below 0.25 ms for every parameter set. The assembly NTT reduces ML-DSA key generation by 10.6 to 13.5% over the C reference (Section 4.6).

Falcon and HAWK build their keys by solving an NTRU lattice equation that can fail and restart.Falcon-512 averages 1369 ms with a standard deviation of 564 ms (up to 91 solver attempts in a single call). Falcon-1024 is the worst case among the post-quantum schemes: 284 of 1000 iterations exceed the 7.16 s counter limit, and the median over all 1000 runs is 5.8 s. HAWK reduces the cost but keeps high variance, with means of 340 ms (HAWK-512) and 1445.32 ms (HAWK-1024).

The fastest key generation is Ed25519, requiring 5.98 ms. Because RSA key generation requires searching for two large prime numbers using the probabilistic Rabin-Miller algorithm, the execution time exceeds the 7.16 s counter limit for all four variants. RSA key generation in this platform is impractical without hardware acceleration. Because key generation executes offline, none of these costs really affect boot latency.

HAWK has the smallest secret key among the lattice schemes with 184 bytes for HAWK-512 and 360 bytes for HAWK-1024. The stack usage for key generation reaches 95.8 KiB for ML-DSA-87, within the 125 KiB user stack budget

Table 4.2: Key generation statistics per algorithm variant. Times in milliseconds at 300 MHz; key sizes in bytes; worst-case stack depth in KiB (Section 4.1.5).

		Time (ms)			**Size (bytes)**		**Stack**
Algorithm	**Level**	μ	σ	**Tries (max)**	**pk**	**sk**	**(KiB)**
ML-DSA-44 (ref)	2	17.58	0.17	—	1,312	2,560	37.8
ML-DSA-44 (asm)	2	15.21	0.16	—	1,312	2,560	37.8
ML-DSA-65 (ref)	3	30.23	0.04	—	1,952	4,032	59.8
ML-DSA-65 (asm)	3	26.62	0.04	—	1,952	4,032	59.8
ML-DSA-87 (ref)	5	50.85	0.22	—	2,592	4,896	95.8
ML-DSA-87 (asm)	5	45.47	0.21	—	2,592	4,896	95.8
Falcon-512	1	1,369.07	563.95	10.43 (91)	897	1,281	17.4
Falcon-1024[1]	5	5,442.91	734.90	18.36 (143)	1,793	2,305	33.6
HAWK-512	1	340.01	111.61	1.10 (3)	1,024	184	14.2
HAWK-1024	5	1,445.32	551.85	1.02 (3)	2,440	360	27.2
Ed25519	—	5.98	0.04	—	32	64	1.6
RSA-2048[2]	—	OVF (> 7.16 s)			256	640	2.2
RSA-3072[2]	—	OVF (> 7.16 s)			384	960	2.2

OVF marks a counter overflow ($> 2^{31} - 1$ cycles, ≈ 7.16 s).

[1]Falcon-1024 μ and σ are computed over the 716 of 1000 runs that did not overflow; the other 284 (28.4%) exceeded the counter, so μ underestimates the true cost. The median over all 1000 runs is 5.8 s.

[2]RSA rows cover both $e = 3$ and $e = 65537$: the key generation cost does not depend on the public exponent. The secret key is the BearSSL CRT representation p, q, d_p, d_q, q^{-1}.

(Section 4.1.5). The reference and assembly ML-DSA builds have identical stack usage; the assembly NTT caches the zeta values in address registers and does not have effect on the stack usage.

Figure 4.1 plots the same data on a logarithmic scale, which spans the three orders of magnitude from Ed25519 (6 ms) to the RSA overflow ceiling.

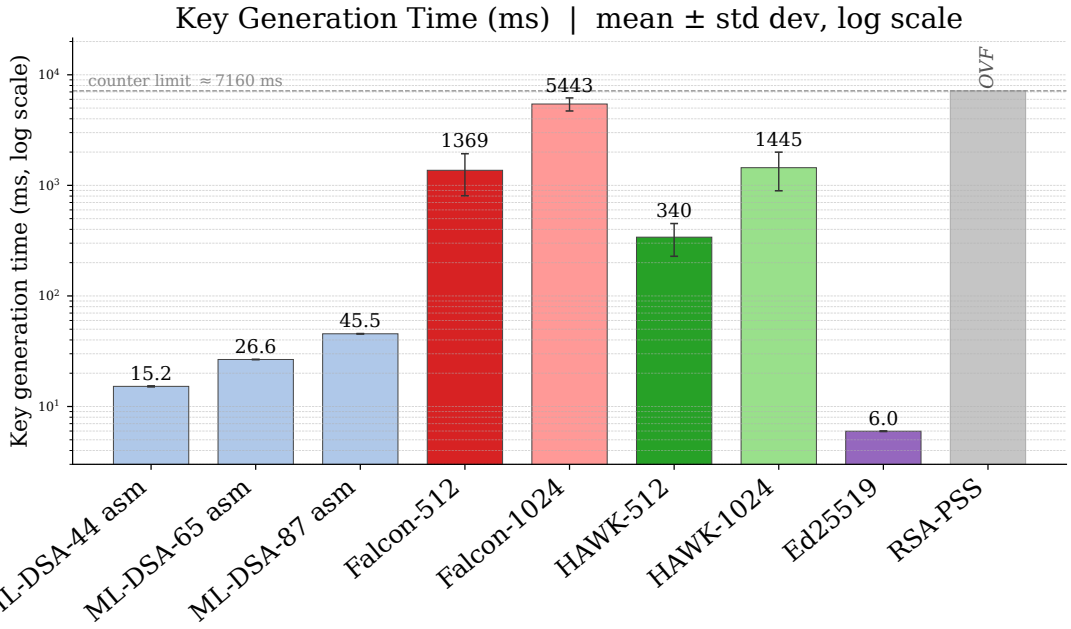


Figure 4.1: Key generation time for all variants (mean ± standard deviation, log scale). ML-DSA is quasi-deterministic (error bars invisible at this scale); Falcon and HAWK show large variance from NTRU-solve restarts. RSA overflows the ≈7160 ms counter limit for all four variants.

4.4 Signature Generation

The following data characterizes the (offline) signing workload for reference.

Table 4.3 reports mean and worst-case sign times, together with the signature size and the stack depth of the signing call. The sign time is decomposed into hash time (sign_hash) and cryptographic sign time (sign_sign).

ML-DSA signing uses the Fiat-Shamir with aborts paradigm. Each signing attempt generates a masking vector, computes a polynomial commitment, and checks that the resulting signature satisfies a norm bound. When the bound is violated, the attempt is discarded and a new one begins. The expected number

Table 4.3: Signature generation statistics. Times in milliseconds at 300 MHz; signature size in bytes; worst-case stack depth in KiB (Section 4.1.5). sign_σ and $\text{sign}_{\max}$ refer to the crypto phase only. Tries (max) reports mean signing attempts with observed maximum in parentheses. ⋆ RSA signing is constant to within <1 μs across all iterations.

		Time (ms)					sig	Stack
Algorithm	**Level**	**hash**	**$\text{sign}_{\text{mean}}$**	**sign_σ**	**$\text{sign}_{\max}$**	**Tries (max)**	**(bytes)**	**(KiB)**
ML-DSA-44 (ref)	2	54.49	67.28	42.44	337.29	4.31 (28)	2,420	51.1
ML-DSA-44 (asm)	2	52.76	42.72	26.12	209.46	4.31 (28)	2,420	51.1
ML-DSA-65 (ref)	3	54.59	111.53	72.54	561.29	5.29 (33)	3,309	78.1
ML-DSA-65 (asm)	3	52.78	68.98	41.98	325.54	5.29 (33)	3,309	78.1
ML-DSA-87 (ref)	5	54.49	137.70	76.40	565.27	3.82 (22)	4,627	120.1
ML-DSA-87 (asm)	5	52.76	90.68	45.43	340.53	3.82 (22)	4,627	120.1
Falcon-512	1	54.92	356.89	0.74	359.46	1.00 (1)	666	41.6
Falcon-1024	5	58.43	1,094.13	1.81	1,101.50	1.00 (1)	1,280	81.3
HAWK-512	1	54.89	14.97	0.04	15.02	1.00 (1)	555	4.7
HAWK-1024	5	54.20	33.86	0.04	33.90	1.00 (1)	1,221	7.7
Ed25519	—	28.67	34.78	<0.01	34.78	—	64	2.0
RSA-2048[1]	—	35.96	1,460.36	constant⋆		—	256	2.4
RSA-3072[1]	—	35.96	5,156.95	constant⋆		—	384	2.4

[1]RSA rows cover both $e = 3$ and $e = 65537$: the signing cost depends on the private exponent only.

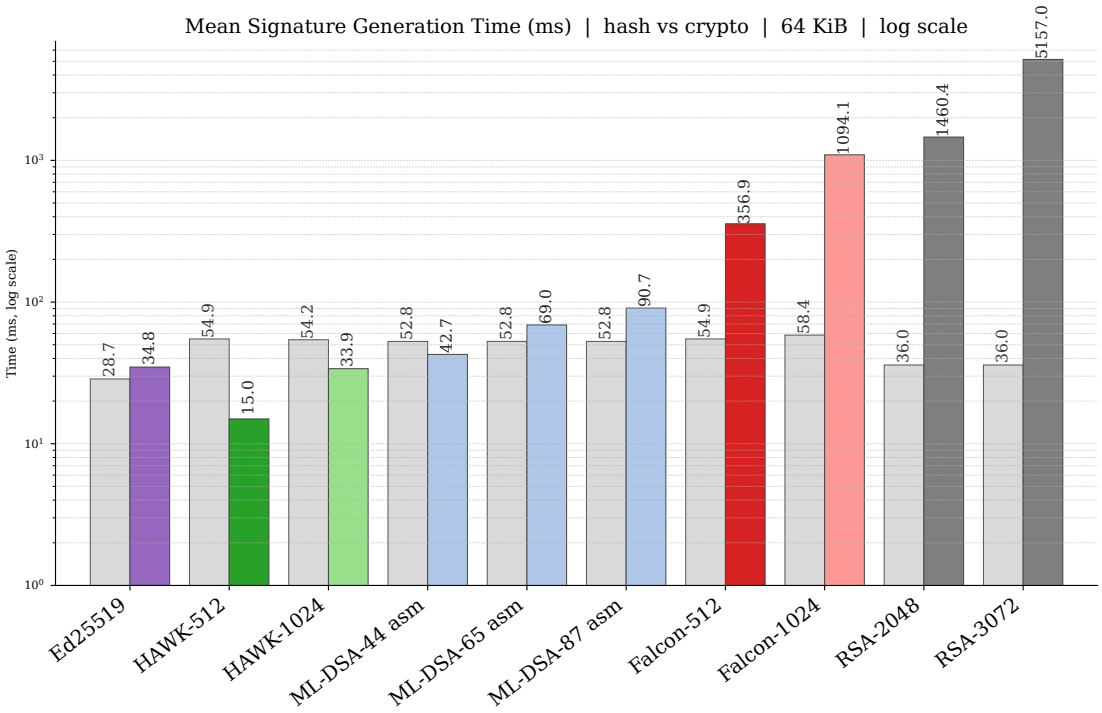


Figure 4.2: Mean signature generation time: hash phase (left bar) and cryptographic phase (right bar), log scale. ML-DSA assembly build only; RSA-2048 and RSA-3072 each shown once (timing is identical for $e = 3$ and $e = 65537$). Labels show hash time and sign time in ms.

of attempts is 4.25 for ML-DSA-44, 5.1 for ML-DSA-65, and 3.85 for ML-DSA-87 [31]. The measured means (4.31, 5.29, 3.82) agree with these values. Because the number of attempts follows a geometric distribution, the sign_sign time has high variance. For safety-critical signing workflows, the max observed value (210 ms for ML-DSA-44 asm over 1000 iterations) is more operationally relevant than the mean. Figure 4.3 shows the observed `sign_try` distribution per ML-DSA parameter set.

ML-DSA-87 signature generation uses 120.1 KiB of stack space, leaving 4.9 KiB of user stack. Because signature generation is performed offline on the host workstation, this constraint does not affect the bootloader deployment. If signature generation were executed on-device, the stack allocation for the calling functions above `crypto_sign_signature` would be limited to approximately 4.9 KiB.

The probability that a Falcon-512 signature exceeds the norm bound and triggers a restart is bounded by $\tau_{\text{sig}}^{2n} \cdot e^{n(1-\tau_{\text{sig}}^2)}$ where $\tau_{\text{sig}} = 1.1$ and $n = 512$, giving an upper bound of 4.9×10^{-5} [32]. No signing restart was observed in 1000 iterations for either Falcon variant. HAWK signing restarts at a rate of approximately 1 in 200,000 for HAWK-512 and 1 in 400,000 for HAWK-1024 [35]. No restart was observed in 1000 iterations.

Falcon-512 signing requires 357 ms because the inner loop of the NTRU-tree Gaussian sampler requires 64-bit double-precision floating-point arithmetic. The TC375TP TriCore has no hardware FP64 unit; typically, all double-precision float operations are software emulated by the TASKING compiler. In case of FALCON this was managed by a custom floating-point implementation in C. HAWK avoids this cost: its signing path uses an integer Gaussian sampler and runs without floating-point arithmetic, which makes HAWK-512 signing 23.8× faster than Falcon-512 (14.97 ms vs. 356.89 ms) on the same hardware. HAWK has the smallest sign stack at security level 1 (4.7 KiB for HAWK-512) compared to Falcon-512 (41.6 KiB). Figure 4.2 shows the hash and crypto breakdown per algorithm (log scale; ML-DSA assembly build only). Figure 4.4 compares the reference and assembly ML-DSA builds directly.

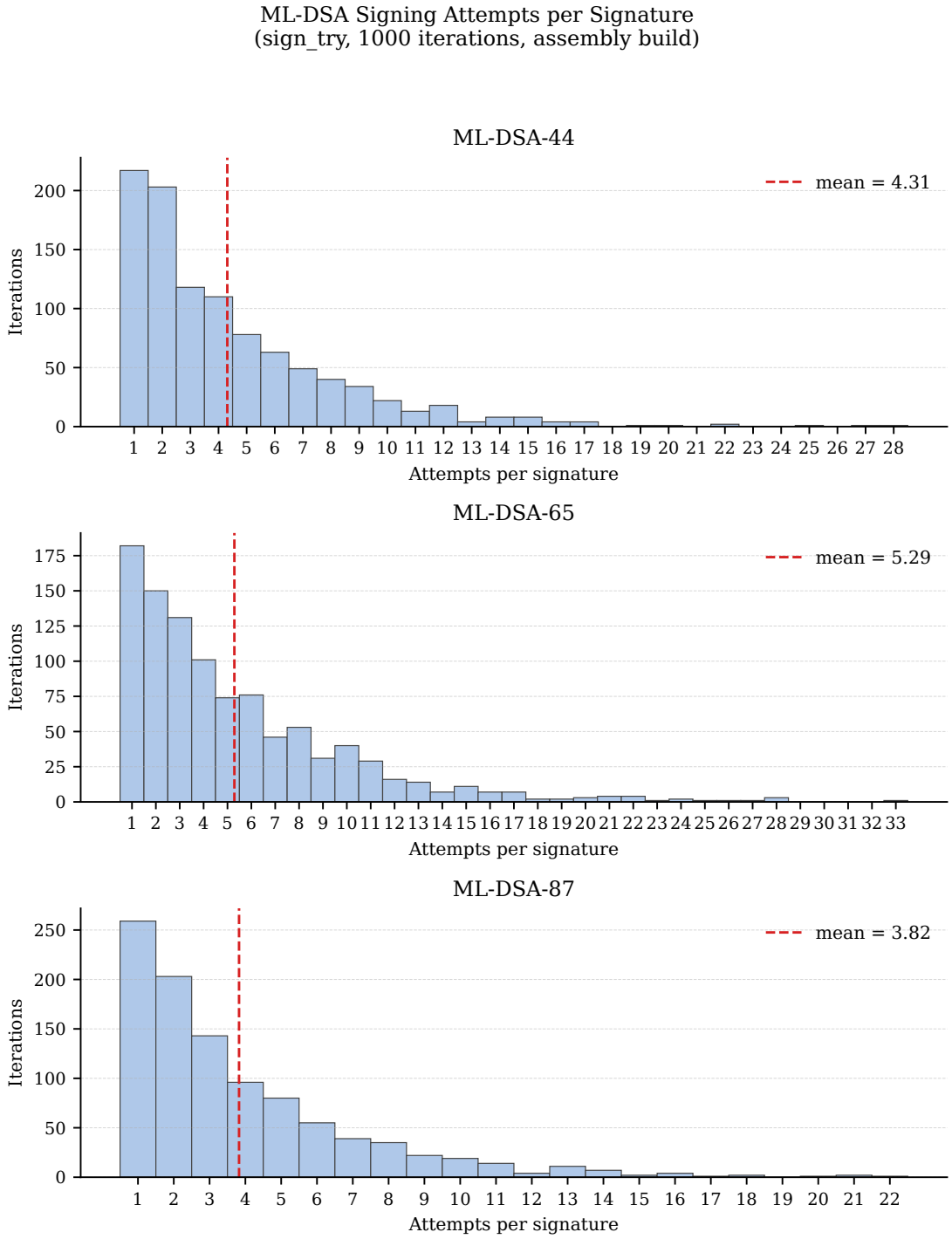


Figure 4.3: Distribution of signing attempts (`sign_try`) per ML-DSA parameter set, 1000 iterations each. A value of 1 means success on the first attempt. The ref and asm builds produce statistically identical distributions for each level (same DRBG seed sequence).

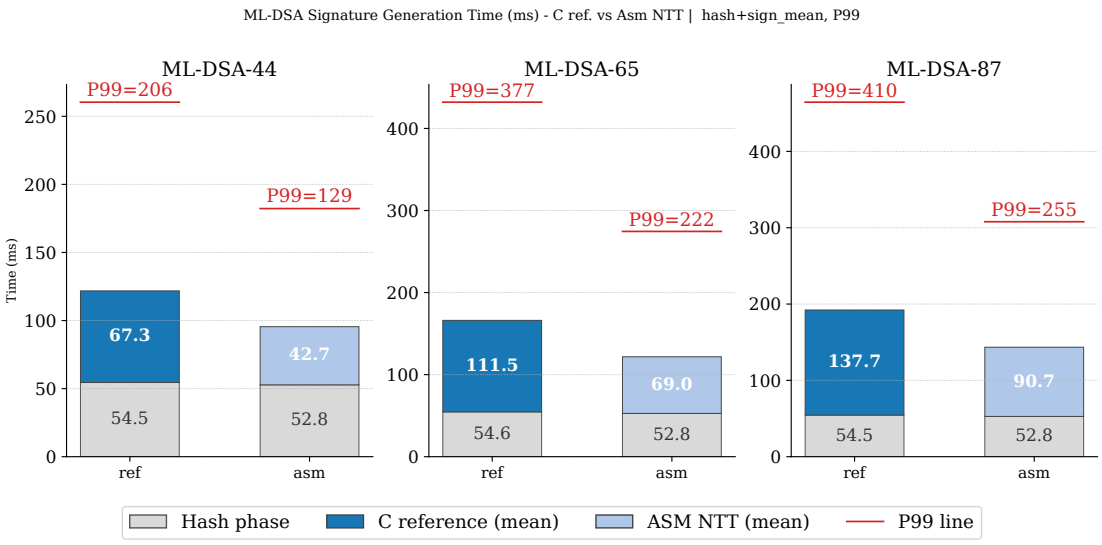


Figure 4.4: ML-DSA signature generation time: C reference vs. assembly NTT, all three parameter sets. Hash phase (lower) and crypto phase (upper). Labels show $\text{sign}_{\text{mean}}$ and sign_{P99} in ms.

4.5 Signature Verification

Signature verification is the most important operation in the bootloader and can be subject to strict timing constraints imposed by the microcontroller's specific operating environment. Table 4.4 reports the mean verification time, decomposed into hash phase and cryptographic phase, together with the combined size of the public key and signature that the macro header must store, the verify stack depth and the flash occupied by the verify path. Verification timing is near-constant for all algorithms.

For all PQC algorithms using SHAKE-256, the hash phase accounts for at least 50% of total verification time; for Falcon-512 this fraction reaches 95.5%. For ML-DSA-87 (asm), where the polynomial verification requires 44.77 ms, the hash phase still accounts for 54.1% of total time.

Ed25519 achieves the lowest total verification time (47.95 ms) because it hashes with SHA-512, faster than SHAKE-256 on this platform (Section 4.2). Ed25519 provides no post-quantum security.

Among post-quantum algorithms, Falcon-512 and HAWK-512 achieve total verification times below 68 ms. ML-DSA-44 (asm) also falls in this range at 67.96 ms. The hash cost is the same for Falcon-512, HAWK-512, and ML-DSA-44 (asm) within measurement noise; only the crypto phase differs.

Falcon-512 has the lowest crypto verification time (2.59 ms); the verification algorithm requires only one lattice projection and a norm check, with no floating-point operations.

The pk+sig column of Table 4.4 reports the combined size of the public key and the signature, the two items the macro header must store. For the secure bootloader, the public key and signature must both fit in the 8 KiB macro header alongside the 96-byte metadata header. The largest combination is ML-DSA-87, with 2,592 bytes for the public key and 4,627 bytes for the signature, for a total of 7,219 bytes. The signature field is padded to the next 32-byte boundary (4,627 to 4,640 bytes); the public key is already a multiple of 32 bytes and receives no padding. Together with the 96-byte metadata header and its 96-byte trailing copy, the header content occupies 7,424 of the 8,192 available bytes, leaving 768 bytes of zero padding.

All used schemes fit within the 125 KiB user stack for every operation. The bootloader executes only `crypto_sign_verify`, so the stack column of Table 4.4 is the operationally relevant one. The worst-case verify stack is ML-DSA-87 at 91.0 KiB, leaving 34.0 KiB of headroom for the bootloader's own call frames above the verify call. This margin is sufficient for any realistic bootloader call depth.

The verify path alone occupies 4.8 to 6.6 KiB of flash for ML-DSA, 6.1 KiB for Falcon, and 31.0 KiB for HAWK, against 9.9 to 12.0, 72, and 104 KiB for the full libraries (Table 4.6). For Falcon, key generation and signing account for over 90%

Table 4.4: Signature verification statistics. Times in milliseconds at 300 MHz; header material in bytes; worst-case stack depth and ROM in KiB (Section 4.1.5). Hash% is the fraction of total verification time spent in message hashing. pk+sig is the sum of the public key and signature sizes.

		Time (ms)				Size (bytes)	Memory (KiB)		
Algorithm	**Level**	**Hash**	**Verif**	**Total**	**Hash %**	**pk+sig**	**Stack**	**ROM verify**	**ROM library**
ML-DSA-44 (ref)	2	54.59	18.92	73.51	74.3	3,732	35.7	4.8	9.9
ML-DSA-44 (asm)	2	52.78	15.18	67.96	77.7	3,732	35.7	6.6	11.7
ML-DSA-65 (ref)	3	54.59	31.12	85.71	63.7	5,261	56.7	4.8	9.9
ML-DSA-65 (asm)	3	52.78	25.33	78.11	67.6	5,261	56.7	6.5	11.6
ML-DSA-87 (ref)	5	54.59	52.88	107.47	50.8	7,219	91.0	4.8	10.2
ML-DSA-87 (asm)	5	52.78	44.77	97.54	54.1	7,219	91.0	6.5	12.0
Falcon-512	1	54.88	2.59	57.47	95.5	1,563	4.7	6.1	72.4
Falcon-1024	5	58.43	8.27	66.71	87.6	3,073	8.3	6.1	71.7
HAWK-512	1	54.89	12.96	67.84	80.9	1,579	6.4	31.0	104.2
HAWK-1024	5	54.22	28.49	82.71	65.6	3,661	11.4	31.0	104.2
Ed25519	—	28.67	19.28	47.95	59.8	96	3.7	46.7	95.6
RSA-2048 ($e = 3$)	—	35.96	33.36	69.32	51.9	512	2.8	—	13.1
RSA-2048 ($e = 65537$)	—	35.96	80.78	116.74	30.8	512	2.8	—	13.1
RSA-3072 ($e = 3$)	—	35.96	81.17	117.13	30.7	768	2.8	—	13.1
RSA-3072 ($e = 65537$)	—	35.96	219.79	255.75	14.1	768	2.8	—	13.1

Note: Sub-components may not sum exactly to the reported totals due to rounding.

ROM verify: object files of the scheme that remain linked when the application calls only `crypto_sign_verify`, with the same object-file scope as Table 4.6. ROM library: full library, from Table 4.6.

of the library: the floating-point FFT code with its 16 KiB twiddle table and the key generation tables are not linked.

Figure 4.5 summarizes the full decomposition (ML-DSA assembly build only; sorted by total). Figure 4.6 compares reference and assembly ML-DSA builds.

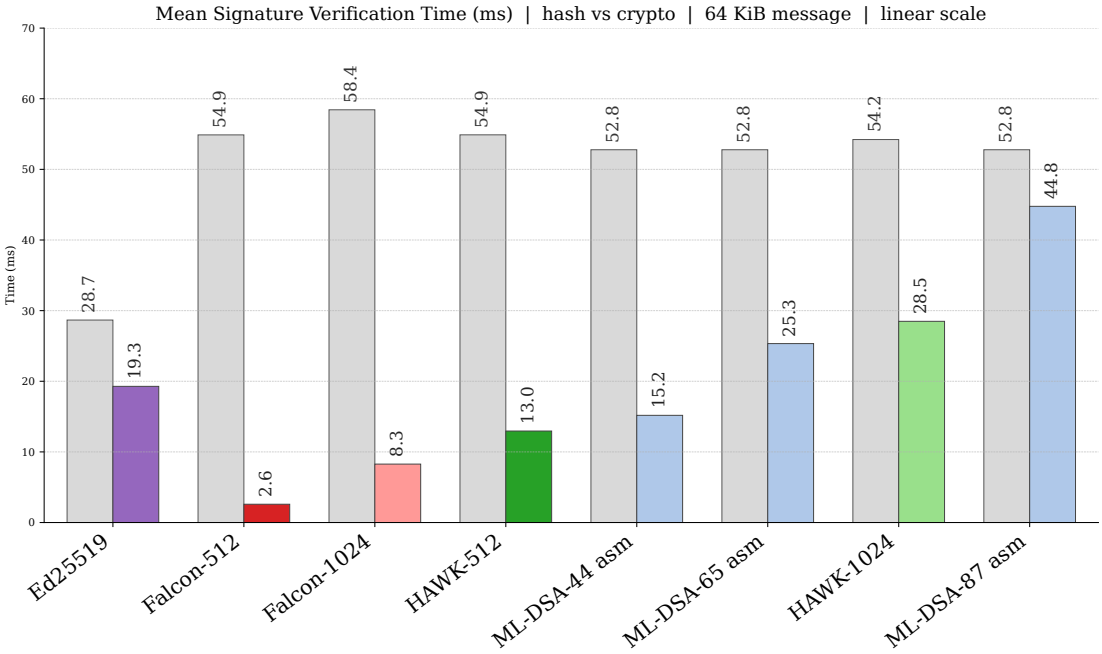


Figure 4.5: Mean signature verification time: hash phase (left bar) and cryptographic phase (right bar), linear scale. ML-DSA assembly build only. All RSA variants are omitted for scale. Labels show hash time and crypto time in ms.

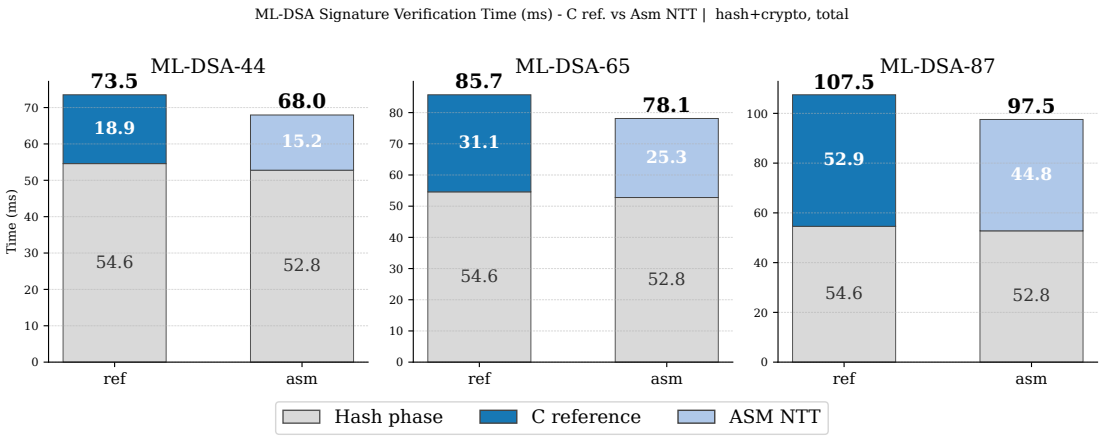


Figure 4.6: ML-DSA signature verification time: C reference vs. assembly NTT, all three parameter sets. Hash phase (lower) and crypto phase (upper).

4.6 NTT Optimization Impact

The TriCore assembly implementation of the NTT is described in Section 3.3.8. Table 4.5 quantifies the performance impact across all three ML-DSA parameter sets. The reference and assembly builds are otherwise identical; only the NTT and INTT functions differ.

Table 4.5: ML-DSA reference vs. assembly NTT: mean crypto timings. All times in milliseconds. Delta% is the relative improvement of the asm variant over the reference.

	Keygen		**Verif (crypto)**		**Verif (total)**	
Level	ref	asm	ref	asm	ref	asm
ML-DSA-44	17.58	15.21 (−13.5%)	18.92	15.18 (−19.8%)	73.51	67.96 (−7.6%)
ML-DSA-65	30.23	26.62 (−11.9%)	31.12	25.33 (−18.6%)	85.71	78.11 (−8.9%)
ML-DSA-87	50.85	45.47 (−10.6%)	52.88	44.77 (−15.3%)	107.47	97.54 (−9.2%)

The isolated NTT and INTT achieve a 94.1% cycle reduction (260,570 to 15,339 cycles for the forward transform). The in-context improvement on total verification time is 7.6–9.2%, because the NTT is one of several operations in the verify path and the hash phase (52–55 ms, unchanged) accounts for over half of total verification time. The improvement on the crypto phase alone (key generation and verif_verif) is 10.6 to 19.8%, a range consistent with the NTT's contribution to those operations.

Both builds are compiled with the same TASKING compiler with the same optimization level (O3). The C reference executes the eight butterfly layers sequentially: the outer loop iterates over the layer distance (128 halving down to 1) and each iteration performs one full pass over the 256-coefficient array, so every coefficient is read from memory and written back at each layer. The assembly port merges the layers in a 3-3-2 structure (Section 3.3.8): each coefficient is read and written three times instead of eight, eight coefficients remain resident in data registers across each merged group, and seven zeta values are cached in address registers.

4.7 ROM Footprint

ROM footprint (flash) determines whether the algorithm fits alongside the bootloader code in the available flash memory. The stack usage of each operation is reported in Tables 4.2, 4.3 and 4.4.

Table 4.6 reports the ROM size of the cryptographic library code only (Section 4.1.5).

Table 4.6: ROM footprint of cryptographic library object files only.

Algorithm	**.text (bytes)**	**.rodata (bytes)**	**Total (KiB)**
ML-DSA-44 (ref)	9,146	1,024	9.9
ML-DSA-44 (asm)	8,896	3,064	11.7
ML-DSA-65 (ref)	9,110	1,024	9.9
ML-DSA-65 (asm)	8,860	3,064	11.6
ML-DSA-87 (ref)	9,470	1,024	10.2
ML-DSA-87 (asm)	9,220	3,064	12.0
Falcon-512	44,202	29,934	72.4
Falcon-1024	43,476	29,982	71.7
HAWK-512	58,456	48,250	104.2
HAWK-1024	58,456	48,250	104.2
Ed25519	65,212	32,632	95.6
RSA-PSS (all)	12,812	588	13.1

ML-DSA has the smallest ROM footprint among the post-quantum algorithms (9.9–12.0 KiB). The PQClean C code is nearly identical across the three parameter sets; the k and ℓ dimension parameters are compile-time constants. The assembly variant adds 2,040 bytes of precomputed zeta values tables in `.rodata` and reduces `.text` by approximately 250 bytes, for a net ROM increase of 1.8 KiB and a 15.3 to 19.8% improvement in crypto verification speed.

Falcon carries approximately 30 KiB of `.rodata` for the precomputed floating-point FFT twiddle table (16 KiB) and the key generation tables. HAWK-512 and HAWK-1024 share identical ROM sizes because the NIST submission reference code uses preprocessor constants to parameterize both variants from the same object files.

Figure 4.7 shows the `.text` and `.rodata` split per algorithm.

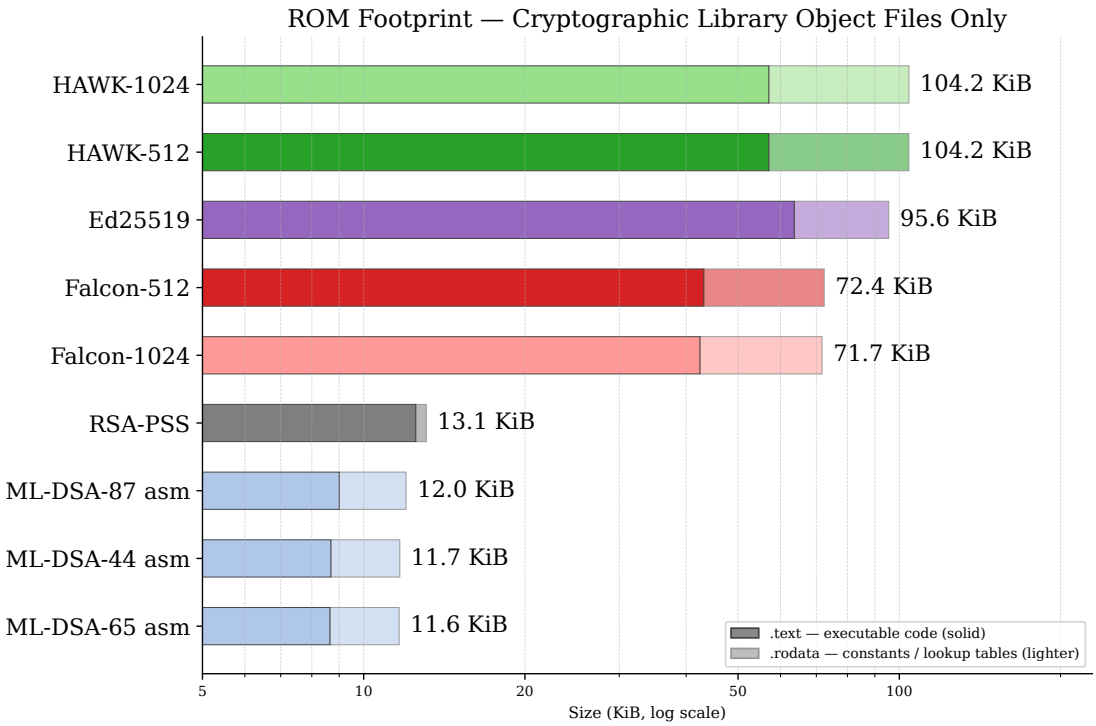


Figure 4.7: ROM footprint of cryptographic library object files per algorithm, split into `.text` (executable code, solid) and `.rodata` (read-only constants and lookup tables, lighter shade).

4.8 Discussion

4.8.1 Hash Dominance

The hash phase dominates verification for every post-quantum scheme that uses SHAKE-256 (Section 4.5). The hash time is identical regardless of the signature scheme used (approximately 54.59 ms for SHAKE-256 in ML-DSA, 52.78 ms for the ML-DSA asm build, 28.67 ms for SHA-512). Halving the firmware image size approximately halves the hash cost and reduces total Falcon-512 verification from 57.47 ms to 30 ms. A hardware Keccak accelerator, if available, would provide a larger improvement than any optimization to the signature algorithm itself.

4.8.2 Comparison at Matched Security Levels

The evaluated schemes do not populate the five NIST security levels evenly. Two schemes meet at level 1 (Falcon-512 and HAWK-512), and three meet at level 5 (ML-DSA-87, Falcon-1024, and HAWK-1024). ML-DSA-44 and ML-DSA-65 stand alone at levels 2 and 3, because no other evaluated scheme offers a variant at those levels, and no evaluated scheme targets level 4. A direct same-level comparison is therefore possible only at levels 1 and 5. The ML-DSA parameter sets are compared separately as one family across levels 2, 3, and 5.

For the bootloader verify path, Falcon-512 is faster (57.47 ms vs. 67.84 ms, Table 4.4), because its verification requires only one lattice norm check (2.59 ms), against HAWK-512's polynomial verification (12.96 ms). Both verification times are dominated by the hash phase, and differ by less than 11 ms in total.

HAWK-512 has a smaller signature (555 vs. 666 bytes, Table 4.3), a larger public key (1,024 vs. 897 bytes, Table 4.2), and 23.8× faster signing (14.97 vs. 356.89 ms). The HAWK-512 sign stack (4.7 KiB) is 8.8× smaller than Falcon-512 (41.6 KiB). Falcon-512 has a smaller verify stack (4.7 vs. 6.4 KiB), and a lower ROM footprint (72 vs. 104 KiB, Table 4.6).

At level 5, Falcon-1024 has the fastest verification (66.7 ms total, 8.3 ms without the hashing phase). Falcon-1024 key generation measurements are unreliable on the TC375TP: 28.4% of iterations overflow the 7.16-second counter limit (Table 4.2). The NTRU equation solver requires an average of 18.36 attempts for Falcon-1024, against 1.02 for HAWK-1024.

HAWK-1024 achieves 82.7 ms verification, 33.9 ms signing, and key generation that completes within the counter range (mean 1,445 ms with no counter overflows).

ML-DSA-87 (asm) has the smallest ROM footprint (12.0 KiB, Table 4.6) but the largest verify stack (91.0 KiB) at this level. Its signature (4,627 bytes) is also the largest, three to four times larger than Falcon-1024 (1,280 bytes) or HAWK-1024 (1,221 bytes). ML-DSA-87 verification (97.54 ms) is the slowest of the three at this

level.

ML-DSA is the only evaluated scheme that spans more than two security levels, so its three parameter sets show how cost grows with the level on a single design. In the assembly build, verification time grows from 67.96 ms at level 2 to 97.54 ms at level 5, and the verify stack grows from 35.7 to 91.0 KiB (Table 4.4). The read-only footprint stays near 12 KiB across all three sets, because the parameter sets share the same code, and differ only in compile-time dimensions. The signature grows from 2,420 to 4,627 bytes, which stays within the 8 KiB header budget at every level (Section 4.5).

4.8.3 Algorithm Selection for the Bootloader

The bootloader executes only verification. Key generation and signing are offline operations running on the development host. The metrics relevant for algorithm selection are then verification latency, signature size (space required in header), ROM footprint, and verify stack depth.

Ed25519 is the fastest to verify but provides no post-quantum security.

At security level 1 neither candidate dominates: Falcon-512 verifies faster and with less stack, HAWK-512 has the smaller signature and the faster signing (Section 4.8.2).

ML-DSA-44 (asm) achieves 67.96 ms total verification at NIST level 2, with the smallest ROM footprint (11.7 KiB) of any post-quantum scheme and standardized status under FIPS 204.

For deployments requiring FIPS standardization, ML-DSA-44 (asm) is the primary candidate. For deployments prioritizing minimal latency and signature size without FIPS requirements, Falcon-512 and HAWK-512 are the primary candidates, with the use of HAWK subject to its standardization as a round-two candidate.

4.8.4 Comparison with Other Work

Previous works report post-quantum cryptography on the same class of automotive microcontroller. Fritzmann et al. implemented key-encapsulation mechanisms on the AURIX TC297TF, a member of the same family, but did not evaluate signature schemes [20]. Bos et al. implemented a Dilithium-based secure boot on the NXP S32G274A, a high-performance vehicle network processor rather than a real-time microcontroller [23]. The most directly comparable is the work of Oka and Jungklass, who measured the verification time of ML-DSA, FN-DSA, and SLH-DSA on the AURIX TC397XX and TC277TF [14]. The TC397XX runs the same TriCore 1.6.2 core at the same 300 MHz clock as the target TC375TP, so its verification times are the direct external reference for the present results.

The comparison evaluates only the cryptographic verification phase. Both isolate the message hashing phase (Section 4.5), ensuring the reported verification figures represent the same operations. Table 4.7 lists the metrics for the five algorithms common to both evaluations.

Table 4.7: Cryptographic verification time on the TriCore 1.6.2 at 300 MHz: this work on the TC375TP versus Oka and Jungklass on the TC397XX [14]. Times in milliseconds. Oka and Jungklass report two memory configurations (see text). Their labels ML-DSA 2/3/5 correspond to ML-DSA-44/65/87, and FN-DSA to Falcon.

Algorithm	**This work**	**Oka (local)**	**Oka (global)**
ML-DSA-44 (asm)	15.18	12.3	41.5
ML-DSA-65 (asm)	25.33	20.7	69.4
ML-DSA-87 (asm)	44.77	34.9	117.6
Falcon-512	2.59	3.5	11.2
Falcon-1024	8.27	7.2	23.0

The three columns measure three different memory placements. This work executes code from program flash with the program cache disabled and keeps the working data in the Data Scratch-Pad RAM (DSPR). The Oka local configuration places both code and data in the core-local scratchpad: code in the Program Scratch-Pad RAM (PSPR) and data in the DSPR. The Oka global configuration executes code from flash with the cache enabled and keeps data in global RAM. The local configuration removes the flash instruction-fetch cost; the global configuration pays it but recovers part of it through the cache.

Against the global configuration, the present results are 2.6 to 2.7 times faster for ML-DSA and 2.8 to 4.3 times faster for Falcon, even though the present work runs with the cache disabled. The optimized Number Theoretic Transform (NTT) and the placement of data in the DSPR outweigh the cache that the global configuration uses. Against the local configuration, the ML-DSA results are 20 to 28 percent slower. This gap is the cost of the stricter, more deployment-realistic setup used here; executing the optimized code from the PSPR would close it.

For Falcon-512, this work measures a verification time of 2.59 ms. This is 0.91 ms faster than Oka's local configuration, which contradicts the memory penalty observed for ML-DSA. The absolute difference is less than one millisecond, but it represents a 26% relative gap. Because Oka and Jungklass do not document their compiler optimization flags, measurement overhead, or whether their figures are averages, the exact cause of this gap cannot be isolated. Given these unknown

variables, the Falcon figures confirm that both implementations execute in the low single-digit milliseconds, but a direct architectural comparison cannot be drawn.

Two limits apply to the whole comparison. The two microcontrollers share the TriCore 1.6.2 core and clock but differ in core count and memory layout, and the memory placements differ as described above, so the columns are not measured under identical conditions. Oka and Jungklass also report single values without an iteration count or a stated measurement instrument, whereas the present figures are means over 1000 iterations obtained from the `CCNT` performance counter. The comparison is therefore indicative rather than exact. Within these limits, the present measurements agree with the external reference and improve on its flash-resident configuration, and the assembly NTT applied here is the type of target-specific optimization that Oka and Jungklass list as future work.

Chapter 5

Conclusion and Future Work

The integrity and authenticity of the firmware a vehicle runs is the basis of its security. The established way to guarantee both is a digital signature via asymmetric cryptography over the firmware image, verified at boot against a trusted public key. A cryptographically relevant quantum computer breaks today's primitives, and expert surveys place such a machine about a decade away or more [8, 57]. This puts the automotive trust chain at risk.

This thesis addresses that threat on an automotive oriented system on a chip. It implements a post-quantum secure bootloader on an Infineon AURIX TC3xx microcontroller, using proposed NIST PQC candidates lattice-based digital signatures and evaluates such schemes on the target hardware for the metrics that matter in an embedded system: verification latency, memory, and code size (Chapter 4).

The evaluation answers two questions: whether post-quantum verification is feasible on this class of device, and what the migration from classical signatures costs at the system level. Post-quantum verification is feasible in software, without dedicated acceleration. Because hashing time scales linearly with image size, the dominant cost for realistic firmware is hashing the application, not the signature operation itself.

5.1 Summary of the Work

The implementation provides a second-stage secure bootloader for the AURIX TC375TP. It employs a dual-slot, copy-on-update firmware policy, an 8 KiB signed metadata header that carries the public key and signature in a fixed layout, and an Ethernet firmware-upload channel built on the lwIP stack. The design is detailed in Chapter 3.

Twelve signature configurations were integrated behind a single detached-signature interface: ML-DSA-44, ML-DSA-65, and ML-DSA-87; Falcon-512 and

Falcon-1024; HAWK-512 and HAWK-1024; the classical baseline Ed25519; and RSA-PSS at 2048 and 3072 bits with public exponents $e = 3$ and $e = 65537$. A host-side tool performs offline signing and trust-anchor provisioning. The forward and inverse NTT of ML-DSA were reimplemented in TriCore assembly, adapted from the ARM Cortex-M4 pqm4 zeta-caching strategy by remapping the cached factors to TriCore address registers [55].

The evaluation benchmarked all configurations on the target silicon for execution time, rejection behaviour, peak stack, code size, and key and signature size, under a uniform methodology: program cache disabled, all temporaries in Data Scratch-Pad RAM (DSPR), and a 64 KiB test message. Chapter 4 reports the full results.

5.2 Principal Findings

5.2.1 Feasibility of Post-Quantum Verification on the Target

Every post-quantum scheme completes verification of the 64 KiB message in under 100 ms. At security level 1, Falcon-512 verifies in 57.47 ms and HAWK-512 in 67.84 ms; ML-DSA-44 in the assembly build verifies in 67.96 ms. The slowest post-quantum configuration is ML-DSA-87 in the assembly build at 97.54 ms (Table 4.4). The classical baseline Ed25519 is faster at 47.95 ms but provides no post-quantum security.

The acceptable verification latency depends on the deployment environment. An electronic control unit on a vehicle network must respond to bus traffic within a window that typically ranges from 30 to 100 ms of the wake-up event, or risk missing critical messages [1, 12]. Controllers with safety or hard real-time functions are held to a tighter target of about 10 ms (Section 1.1). The measured post-quantum verification times, from 57.47 to 97.54 ms over the 64 KiB image, fall within the 100 ms upper limit but not within the 30 ms one, and exceed the 10 ms target by over 5 times. An ECU bound by a 30 ms or a 10 ms budget therefore cannot use any of the evaluated schemes at this image size. The message hash alone takes 52.78 to 58.43 ms for the SHAKE-256 schemes, 28.67 ms for Ed25519 and 35.96 ms for RSA-PSS, so at 64 KiB no software configuration, classical baselines included, meets either the 30 ms or the 10 ms target (Section 5.2.2). Verification latency is deterministic across iterations for every scheme, causing the mean and worst case to coincide.Bos et al. [23] and Oka and Jungklass [14] reach the same conclusion on the cryptographic cost of post-quantum secure boot on automotive processors; Section 4.8.4 compares their measurements with the present results.

These figures hold for the 64 KiB benchmark image. The verification latency for larger firmware is set by the hash phase, not by the signature scheme, and is

analysed in Section 5.2.2 and Section 5.4.

The memory cost stays within the platform budget. The worst-case verification stack is ML-DSA-87 at 91.0 KiB, against a 125 KiB user-stack allocation, leaving 34.0 KiB of headroom; every other scheme requires less memory (Table 4.4). The largest public-key and signature pair is ML-DSA-87 at 7,219 bytes, which fits the 8 KiB signed header, together with the two 96-byte metadata headers and the 32-byte field alignment, with 768 bytes to spare (Table 4.4). No configuration exceeds the verification resource budget of the bootloader.

5.2.2 Hash Latency Dominance

For every post-quantum scheme that uses SHAKE-256, the message hash accounts for more than half of total verification time. The fraction reaches 95.5 % for Falcon-512 and remains 54.1 % for ML-DSA-87 in the assembly build, whose cryptographic phase is the heaviest of the post-quantum set (Table 4.4). The hash cost is identical regardless of the signature scheme, because it depends only on the message length and the hash function: approximately 54.59 ms for SHAKE-256 and 28.67 ms for the SHA-512 used by Ed25519. Under the 10 ms target only Falcon-512 and Falcon-1024 fit, at 2.59 and 8.27 ms.

This result reorders the optimization priorities. The signature scheme is not the lever for boot-time verification latency; the firmware size and the hash throughput are. The observation that verification time and on-chip memory, rather than the signature primitive itself, are the binding constraints for post-quantum secure boot matches the requirement analysis of Marzougui and Krämer [15]. Bos et al. reach the same conclusion for Dilithium secure boot on an automotive network processor: without hardware support for SHAKE-256, the image verification is dominated by the hashing of the image [23].

The hash time scales linearly with the firmware size, while the cryptographic phase does not depend on it. Realistic update images for automotive ECUs range from 1.1 to 5.9 MiB [50]. At these sizes the hash phase on this platform takes roughly 1 to 5 seconds, while the cryptographic verification stays between 2.59 and 44.77 ms. At production image sizes the choice of signature scheme therefore has almost no effect on total boot latency. The options that remain are a faster hash or a different boot architecture; both are discussed in Section 5.4.

5.2.3 Limited System-Level Impact of NTT

The TriCore assembly implementation of the NTT reduces the cost of the isolated forward transform by 94.1 % (260,570 to 15,339 cycles). Measured on the operations that contain the transform, the assembly build reduces ML-DSA key generation by

10.6 to 13.5 %, signing by 34.1 to 38.2 % (Table 4.3), and the cryptographic verification phase by 15.3 to 19.8 % (Table 4.5), across the three parameter sets. Signing benefits most because each rejection-sampling attempt repeats the transform. The effect on the full verification time, which also includes the firmware hash, is smaller: total verification decreases by only 7.6 to 9.2 %, because the unchanged hash phase accounts for over half of that time. The optimization therefore reduces the latency of the cryptographic operations themselves, but is bounded at the system level by the dominant hash cost.

5.2.4 Memory Footprint Fits Embedded Deployment

Every configuration fits both the flash and the stack budget. Among post-quantum schemes, ML-DSA has the smallest read-only footprint at 9.9 to 12.0 KiB, since the PQClean C code is nearly identical across parameter sets and the dimensions are compile-time constants (Table 4.6). The assembly build adds 1.8 KiB of precomputed zeta values tables. Falcon carries about 30 KiB of precomputed tables (FFT twiddle table and key generation tables), and HAWK and Ed25519 are the largest at approximately 104 KiB and 96 KiB. The verification stack depth is well below the 125 KiB user budget for all schemes (Table 4.4).

5.2.5 Suitability of the Evaluated Schemes

Among the evaluated schemes, no single one dominates across all metrics (Section 4.8). ML-DSA-44 in the assembly build is the primary candidate where standardization is required, being finalized under FIPS 204 [31] and carrying the smallest post-quantum read-only footprint. Falcon-512 and HAWK-512 are the primary candidates where minimal verification latency and signature size outweigh the requirement for a finalized standard, with the use of HAWK subject to its standardization as a round-two candidate whose security assumption was revised in version 1.1 [35]. ML-DSA remains the conservative choice where long-term assurance is required. The level-1 and level-5 trade-offs are discussed in Section 4.8.2.

5.2.6 Cost of Post-Quantum Migration

The effect of the migration on verification performance depends on whether the firmware hash is counted. On the cryptographic operation alone, the migration carries no penalty on this platform, and in places it is an improvement. The post-quantum cryptographic verification ranges from 2.59 ms for Falcon-512 to 44.77 ms for ML-DSA-87 in the assembly build, against 33.36 ms for RSA-PSS-2048 with $e = 3$ and 80.78 ms with $e = 65537$ (Table 4.4). Falcon-512 is faster even

than the on-chip HSM, whose elliptic-curve accelerator is rated at 100 verifications per second, that is 10 ms per P-256 verification in hardware (Section 3.2.1). The post-quantum schemes therefore match or beat the classical alternatives, including dedicated hardware, on the cryptographic phase, even though they run in software on the main core and the HSM cannot execute them at all for lack of Keccak and NTT acceleration.

Once the firmware hash is included, the migration becomes almost negligible. The hash is the same for every scheme and accounts for more than half of total verification over the 64 KiB image; for production images of several megabytes it accounts for nearly all of it. Total verification of the post-quantum schemes, from 57.47 to 97.54 ms, sits in the same band as RSA-PSS-2048 with $e = 3$ at 69.32 ms. The performance impact of adopting post-quantum secure boot on this microcontroller is therefore set by the firmware size, not by the signature scheme. The migration cost that remains is the larger cryptographic material, which the 8 KiB header absorbs, and the offline signing time, which never enters boot latency.

An independent evaluation by La Manna et al. reaches the same conclusion on a different automotive platform [50]. They measure Falcon and Dilithium verification on a Xilinx Zynq UltraScale+ ZCU102 board across update sizes of 1.1, 2.7, and 5.9 MiB, and report that the verification time grows with the message size and not with the security level, because the software SHAKE permutation dominates the cost.

5.3 Limitations

The prototype was built under development-environment constraints. The following limitations apply. None of them affects the validity of the measurements reported in Chapter 4. The first conditions the security claim rather than the measurements: the chain of trust holds only under the assumption that the bootloader region cannot be rewritten, an assumption a production deployment must enforce in hardware (Section 1.1).

The trust-anchor public key hash is stored alongside with the bootloader in flash memory rather than in a write protected region. Chapter 3 documents this as a development compromise: signing keys changed often during development, and write-protecting the flash region holding each new key hash would have turned every rotation into a flash-management procedure (Sections 3.3.7 and 3.3.11). In production, the key hash is stored in flash and that region is sealed write-protected. Pott et al. demonstrate this OTP locking on the same microcontroller family [51]. The Hardware Security Module (HSM) was not used, for the reasons given in Chapter 3: it lacks Keccak or any useful acceleration and the HSM itself is usable only through a proprietary licensed toolchains.

The verification path processes only public data (the public key, the signature, and the firmware image), so timing side channels disclose no secret on the bootloader. Fault injection remains relevant: the boot decision logic uses sentinel constants separated by a Hamming distance of 16, so that a corrupted status value defaults to a rejection (Section 3.3.6), following the Infineon software fault-mitigation guidance [11]. The cryptographic library code itself carries no such countermeasures, and the construction has not been evaluated against a deliberate glitching adversary. The project code was aligned to MISRA-C guidance only partially, and the integrated third-party libraries were not audited for compliance.

The evaluated set covers the stateless lattice signatures (ML-DSA, Falcon, HAWK) and the classical baselines. It does not include the stateful hash-based signatures LMS and XMSS, which NIST SP 800-208 [16] and CNSA 2.0 [17] recommend for firmware and software signing. These schemes rest on hash-function security alone and place the operational cost on per-key state management at the offline signer, a distinct design point from the stateless schemes evaluated here. Their omission is a limitation of the scheme selection.

Randomness comes from a Deterministic Random Bit Generator (DRBG) built on SHAKE and seeded with a fixed value, this is because the True Random Number Generator (TRNG)is only usable through the proprietary HSM stack. The fixed seed allows to have reproducible benchmarks, and it gives the reference and assembly ML-DSA builds identical rejection sequences, so their timings are directly comparable (Chapter 4). Deterministic generators seeded from a secret value stored in protected flash, combined with a boot counter, are the established software-only practice on this platform class [51]; production deployment would provision a per-device seed in protected storage. Randomness quality affects key generation and signing, both offline operations in a real scope; the verification path consumes no randomness.

Finally, Falcon-1024 key generation is slow and irregular on this platform: 28.4 % of 1000 runs exceeded the 7.16-second measurement window, and the median is 5.8 s (Table 4.2). Key generation runs offline, so this does not affect the bootloader; it only bounds hypothetical on-device use of that scheme.

5.4 Future Work

The benchmark identifies the firmware hash as the primary latency cost. The following directions describe approaches to reduce this cost; the last one addresses deployment rather than performance.

5.4.1 Reducing the Dominant Hash Cost

FIPS 204 defines HashML-DSA, a pre-hash variant of ML-DSA [31]. The verifier still hashes the full firmware, but with a hash function of its choice; the mandated SHAKE-256 then runs only over the short digest. The choice of the message hash is what matters on this platform: SHA-512 processes the 64 KiB message in 28.67 ms against 54.59 ms for SHAKE-256 (Table 4.1), so the pre-hash variant would reduce the dominant cost by half. The pre-hash function must provide collision resistance matching the security level of the parameter set.

The second point is the Keccak permutation itself. Keccak-f[1600] is the inner function of SHAKE-256 and runs for the full duration of the firmware hash, so a TriCore assembly implementation of the permutation, validated bit-for-bit against the C reference, would act on the dominant portion of verification. The permutation accounts for more than half of verification time, so it is the term worth attacking, unlike the cryptographic arithmetic already optimized in Section 5.2.3.

A hardware unit for hash accelleration would give the largest single gain. On the same microcontroller family, the hardware SHA-256 accelerator of the HSM runs about 4.5 times faster than the software implementation on the main core [51]. Applied to the 35.96 ms SHA-256 time of the 64 KiB message, the accelerator would bring the hash to about 8 ms, which with the 2.59 ms cryptographic phase of Falcon-512 still lands above the 10 ms target A future device with a Keccak unit reachable from the main cores or HSM access without the proprietary toolchain, would shrink the dominant term accordingly.

5.4.2 Parallel Verification and Deferred Enforcement

The TC375TP has three TriCore cores, and the verification path uses one. Restructuring the signed message so that the signature covers only the metadata header, which already carries the firmware hash, would let one core hash the firmware while a second core verifies the signature over the header alone. Total latency then is equal to the maximum of the two tasks instead of their sum. This does not remove the bottleneck: the maximum is the firmware hash, so the saving is capped at the cryptographic phase, between 2.59 and 44.77 ms. At the 64 KiB test size this is up to 45 % for ML-DSA-87, but at production image sizes of 1.1 to 5.9 MiB [50] it falls below 5 %. Parallel verification therefore does not solve the latency problem. It also changes the trust model, since the signature would cover the header that binds the firmware through its hash rather than the firmware bytes directly.

In the deferred method the cores are organized differently. Instead of splitting the verification across cores, core 0 runs the complete verification sequentially, hash and signature, while the application cores already execute the same image; core 0 halts them if the verification fails. This removes the hash from the boot critical path entirely, at a price: the architecture moves from the Protected property

of NIST SP 800-193 to a detect-and-recover property [43]. The scheme was not implemented in this work; it is an option to weigh per electronic control unit, and it holds only under two conditions. The application must produce no irreversible or visible effect before the verification outcome. The application cores must not be able to write core 0's memory or the halt and reset registers, which the bootloader can enforce through the Memory Protection Unit (MPU) before releasing them; Pott et al. demonstrate this core-isolation construction on the same family [51].

5.4.3 Scheme-Specific Transform Optimization

No single NTT serves the three lattice schemes, because each one fixes its own ring, modulus, and roots of unity. ML-DSA uses 256-coefficient polynomials with $q = 8380417$ and the primitive 512th root of unity $\zeta = 1753$ for every parameter set, which is why one assembly port covers all three variants. Falcon uses 512- or 1024-coefficient polynomials with $q = 12289$ and the generator $g = 7$, a primitive 2048th root of unity modulo q. HAWK works over the exact integers in $\mathbb{Z}[x]/(x^n+1)$ rather than modulo a fixed prime, and computes its NTT modulo one small prime when signing, and modulo two larger primes when verifying.

Falcon verification uses only integer arithmetic; the floating-point cost that penalizes it on the TriCore sits in signing and key generation, which run offline on the host. An optimized NTT therefore speeds up exactly the operation the bootloader executes. The modulus $q = 12289$ fits in 16 bits, and the TriCore provides packed half-word instructions that operate on two 16-bit values in one 32-bit register, so a packed NTT could process two coefficients per instruction. The ML-DSA modulus needs 23 bits and excludes this packing, which is why the existing assembly port uses one 32-bit coefficient per register. Whether a similar treatment applies to the HAWK transforms is left to a dedicated evaluation, together with a level-matched comparison between HAWK and Falcon.

5.4.4 Key Management and the Update Channel

This implementation trusts one fixed public-key hash per supported algorithm, compiled into the bootloader as its root of trust (Section 3.3.7). The bootloader cannot replace one of these anchors, and it cannot revoke a compromised key. A production deployment needs both. This requires the trust-anchor table to be updatable, and the table lives inside the bootloader, so updating it means updating the bootloader itself. Key management on this platform is therefore tied to the security of the bootloader-update path.

The current update channel authenticates the firmware but not its delivery. The bootloader verifies the signature of an image before it boots, so an unsigned or tampered image is never executed (Chapter 3). The upload path itself, however, is

an open TCP server: any host that can reach the interface can write an image into the staging slot (Section 3.3.10). Boot-time verification stops such an image from running, but it does not stop the write, which leaves the channel open to denial of service and to delivery of unauthenticated content. Mutual Transport Layer Security (mTLS) between the vehicle and the manufacturer backend would close this gap. Both endpoints present certificates, so only an authenticated backend can write the staging slot. Post-quantum signatures such as ML-DSA or Falcon in these certificates would keep the channel authentication quantum-resistant.

The firmware verification path is already cryptographically agile: the signature interface is indexed by an algorithm identifier, so a new scheme enters as a module without changes to the bootloader core. When FIPS 206 finalizes Falcon, or further schemes are standardized, integration reduces to adding the module and provisioning a trust anchor for it.

5.5 Closing Remarks

The goal of this thesis was to implement and evaluate a secure bootloader for an automotive microcontroller, built around the post-quantum signature schemes standardized by NIST. The bootloader was developed and tested on the AURIX TC375TP.

The answer separates into two budgets. On the cryptographic budget the migration is feasible: every evaluated post-quantum scheme fits the memory and code-size limits of the platform, runs in software without dedicated acceleration, and on the cryptographic phase alone matches or exceeds the classical baselines. On the end-to-end boot budget no evaluated scheme is feasible, and this is not a consequence of the migration.

Already at the 64 KiB test size every configuration exceeds a 30 ms wake budget and the 10 ms target of safety controllers, the classical baselines included (Table 4.4); at production image sizes of 1.1 to 5.9 MiB [50] the hash phase alone reaches 1 to 5 seconds and leaves the 30 to 100 ms range by more than an order of magnitude. The binding cost is the firmware hash, which is independent of the signature scheme, so meeting a boot budget under the Protected property of NIST SP 800-193 [43] is a hash-throughput and boot-architecture problem rather than a post-quantum one (Section 5.4). The hardening that a production root of trust requires, such as key revocation and a write-protected anchor, is a property of the deployment platform and does not depend on the choice of signature scheme.

Appendix A

Clock Cycle Benchmarks

Table A.1: Key generation per algorithm variant, in million clock cycles (Mcc means 10^6 cycles; at 300 MHz 1 Mcc $\approx$ 3.33 ms). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072). *OVF* marks a counter overflow ($> 2^{31} - 1$ cycles, $\approx$ 7.16 s).

Algorithm variant	μ **(Mcc)**	σ **(Mcc)**	**P99 (Mcc)**	**Tries (max)**
ML-DSA-44 (ref)	5.28	0.05	5.39	—
ML-DSA-44 (asm)	4.56	0.05	4.67	—
ML-DSA-65 (ref)	9.07	0.01	9.09	—
ML-DSA-65 (asm)	7.99	0.01	8.01	—
ML-DSA-87 (ref)	15.25	0.07	15.41	—
ML-DSA-87 (asm)	13.64	0.06	13.79	—
Falcon-512	410.72	169.19	984.43	10.43 (91)
Falcon-1024	1632.87	220.47	—	18.36 (143)
HAWK-512	102.00	33.48	255.36	1.10 (3)
HAWK-1024	433.60	165.55	1083.23	1.02 (3)
Ed25519	1.79	0.01	1.81	—
RSA-2048 ($e = 3$)	OVF (> 7.16 s)			
RSA-2048 ($e = 65537$)	OVF (> 7.16 s)			
RSA-3072 ($e = 3$)	OVF (> 7.16 s)			
RSA-3072 ($e = 65537$)	OVF (> 7.16 s)			

Falcon-1024 statistics cover the 716 of 1000 iterations within the counter limit; the other 284 (28.4%) overflowed. The median over all 1000 runs is 5.8 s. The 99th percentile is not reported: it lies above the counter limit.

Table A.2: Message hashing (signing phase, 64 KiB message) per algorithm variant, in raw clock cycles (cc; at 300 MHz 1 cycle ≈ 3.33 ns). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072).

Algorithm variant	μ **(cc)**	σ **(cc)**	**P99 (cc)**
ML-DSA-44 (ref)	16,347,726	26	16,347,745
ML-DSA-44 (asm)	15,827,870	0	15,827,870
ML-DSA-65 (ref)	16,376,640	0	16,376,640
ML-DSA-65 (asm)	15,833,667	0	15,833,667
ML-DSA-87 (ref)	16,347,741	20	16,347,760
ML-DSA-87 (asm)	15,827,883	0	15,827,883
Falcon-512	16,476,014	0	16,476,014
Falcon-1024	17,530,279	1	17,530,279
HAWK-512	16,466,792	0	16,466,792
HAWK-1024	16,261,016	0	16,261,016
Ed25519	8,599,889	0	8,599,889
RSA-2048 ($e = 3$)	10,787,960	0	10,787,960
RSA-2048 ($e = 65537$)	10,787,971	0	10,787,971
RSA-3072 ($e = 3$)	10,787,960	0	10,787,960
RSA-3072 ($e = 65537$)	10,787,961	0	10,787,961

Hash function differs by scheme: ML-DSA, Falcon and HAWK use Keccak (SHAKE/SHA3); Ed25519 uses SHA-512; RSA-PSS uses SHA-256. All hash the same 64 KiB message.

Table A.3: Signature generation (cryptographic phase) per algorithm variant, in million clock cycles (Mcc means 10^6 cycles; at 300 MHz 1 Mcc $\approx$ 3.33 ms). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072).

Algorithm variant	μ **(Mcc)**	σ **(Mcc)**	**P99 (Mcc)**	**Tries (max)**
ML-DSA-44 (ref)	20.19	12.73	61.76	4.31 (28)
ML-DSA-44 (asm)	12.82	7.84	38.84	4.31 (28)
ML-DSA-65 (ref)	33.46	21.76	113.23	5.29 (33)
ML-DSA-65 (asm)	20.69	12.59	66.52	5.29 (33)
ML-DSA-87 (ref)	41.31	22.92	123.00	3.82 (22)
ML-DSA-87 (asm)	27.20	13.63	76.52	3.82 (22)
Falcon-512	107.07	0.22	107.60	1.00 (1)
Falcon-1024	328.24	0.54	329.44	1.00 (1)
HAWK-512	4.49	0.01	4.50	1.00 (1)
HAWK-1024	10.16	0.01	10.17	1.00 (1)
Ed25519	10.43	0.00	10.43	—
RSA-2048 ($e = 3$)	438.11	0.00	438.11	—
RSA-2048 ($e = 65537$)	438.11	0.00	438.11	—
RSA-3072 ($e = 3$)	1547.09	0.00	1547.09	—
RSA-3072 ($e = 65537$)	1547.09	0.00	1547.09	—

Table A.4: Message hashing (verification phase, 64 KiB message) per algorithm variant, in raw clock cycles (cc; at 300 MHz 1 cycle ≈ 3.33 ns). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072).

Algorithm variant	μ **(cc)**	σ **(cc)**	**P99 (cc)**
ML-DSA-44 (ref)	16,376,614	0	16,376,614
ML-DSA-44 (asm)	15,833,675	0	15,833,675
ML-DSA-65 (ref)	16,376,631	0	16,376,631
ML-DSA-65 (asm)	15,833,676	0	15,833,676
ML-DSA-87 (ref)	16,376,631	0	16,376,631
ML-DSA-87 (asm)	15,833,665	0	15,833,665
Falcon-512	16,464,467	0	16,464,467
Falcon-1024	17,530,283	0	17,530,283
HAWK-512	16,466,774	0	16,466,774
HAWK-1024	16,266,876	0	16,266,876
Ed25519	8,599,885	0	8,599,885
RSA-2048 ($e = 3$)	10,787,966	0	10,787,966
RSA-2048 ($e = 65537$)	10,787,972	3	10,787,978
RSA-3072 ($e = 3$)	10,787,955	0	10,787,955
RSA-3072 ($e = 65537$)	10,787,961	0	10,787,961

Hash function differs by scheme: ML-DSA, Falcon and HAWK use Keccak (SHAKE/SHA3); Ed25519 uses SHA-512; RSA-PSS uses SHA-256. All hash the same 64 KiB message.

Table A.5: Signature verification (cryptographic phase) per algorithm variant, in raw clock cycles (cc; at 300 MHz 1 cycle ≈ 3.33 ns). Statistics over 1000 iterations (10 for RSA-2048, 3 for RSA-3072).

Algorithm variant	μ **(cc)**	σ **(cc)**	**P99 (cc)**
ML-DSA-44 (ref)	5,676,914	142	5,677,387
ML-DSA-44 (asm)	4,553,093	2,059	4,555,288
ML-DSA-65 (ref)	9,336,725	240	9,337,353
ML-DSA-65 (asm)	7,600,375	166	7,600,838
ML-DSA-87 (ref)	15,863,765	270	15,864,616
ML-DSA-87 (asm)	13,429,713	311	13,430,474
Falcon-512	777,584	13,919	791,657
Falcon-1024	2,482,323	13,045	2,496,828
HAWK-512	3,886,708	653	3,888,573
HAWK-1024	8,546,239	663	8,547,061
Ed25519	5,784,187	46,231	5,882,349
RSA-2048 ($e = 3$)	10,006,643	625	10,006,939
RSA-2048 ($e = 65537$)	24,234,748	625	24,235,044
RSA-3072 ($e = 3$)	24,351,788	1,443	24,353,405
RSA-3072 ($e = 65537$)	65,937,676	1,444	65,939,294

Appendix B

Signer

B.1 Firmware Signer Script (`sign.py`)

Python script used for signing and packaging firmware update images for the secure bootloader.

Listing B.1: Python script for firmware signing and packaging (`sign.py`)

```
import intelhex
import os
import hashlib
import pyqcsign as libpyqcsign

ALGO_IDS = {'mldsa44': 1, 'ed25519': 2, 'falcon512': 3, 'falcon1024':
    4, 'hawk512': 5, 'hawk1024': 6, 'rsa2048pss': 7, 'rsa3072pss': 8,
    'mldsa65': 9, 'mldsa87': 10}


HASH_SHA3_256 = 3
HEADER_SIZE = 96
MACRO_HEADER_SIZE = 8192
MAX_BIN_SIZE = int(1.5 * 1024 * 1024)


def get_algo_sizes(algo_name): #  algoname ->f()-> tupla con sizes
    algo_pk_sz = getattr(libpyqcsign, f"{algo_name.upper()}
    _PUBLICKEYBYTES")
    algo_sk_sz = getattr(libpyqcsign, f"{algo_name.upper()}
    _SECRETKEYBYTES")
    algo_sig_sz = getattr(libpyqcsign, f"{algo_name.upper()}_BYTES")
    return algo_pk_sz, algo_sk_sz, algo_sig_sz

def build_metadata(*, binapp_len, v_maj, v_min, v_pat, algo_id,
    hash_id, algo_pk_sz, algo_sig_sz, binapp_hash, core_pcs):
```

```
22
23      buf = bytearray(HEADER_SIZE)
24
25      buf[0:4] = b'AURX'
26      buf[4:8] = (0).to_bytes(4, 'little')
27      buf[8:12] = binapp_len.to_bytes(4, 'little')
28      buf[12] = HEADER_SIZE
29      buf[13] = 0
30      buf[14:16] = (0).to_bytes(2, 'little')
31      buf[16] = v_maj
32      buf[17] = v_min
33      buf[18] = v_pat
34      buf[19] = algo_id
35      buf[20] = hash_id
36      buf[21] = 0
37      buf[22:24] = algo_pk_sz.to_bytes(2, 'little')
38      buf[24:26] = algo_sig_sz.to_bytes(2, 'little')
39      buf[26:32] = b'\x00' * 6
40      for i in range(6):
41          base = 32 + (i * 4)
42          buf[base:base+4] = core_pcs[i].to_bytes(4, 'little')
43
44      buf[56:64] = b'\x00' * 8
45      buf[64:96] = binapp_hash
46
47      return bytes(buf)
48
49  # funzione supporto per il dir delle keys
50  def _resolve_key_path(path: str, keys_dir: str = "keys") -> str:
51      if os.path.exists(path):
52          return path
53      candidate = os.path.join(keys_dir, path)
54      if os.path.exists(candidate):
55          return candidate
56      return path
57
58  def _write_rot_c(algo, pk_sha3):
59      rot_file = f"{algo}_rot.c"
60      with open(rot_file, "w") as f:
61          f.write(f"/* rot key material for {algo} */\n")
62          f.write("#include <stdint.h>\n\n")
63          f.write(f"const uint8_t {algo}_pubkey_sha3_256[32] = {{\n")
64          lines = [", ".join([f"0x{b:02X}" for b in pk_sha3[i:i+8]])
        for i in range(0, len(pk_sha3), 8)]
65          f.write("    " + ",\n    ".join(lines))
66          f.write("\n};\n")
67      print(f"[>] created {rot_file} file for {algo}.")
68
69  def _do_keygen_logic(algo, keys_dir="keys"):
```

```
70      os.makedirs(keys_dir, exist_ok=True)
71      algo_pk_sz, algo_sk_sz, algo_sig_sz = get_algo_sizes(algo)
72      keypair_fn = getattr(libpyqcsign, f"{algo}_keypair")
73      pk, sk = keypair_fn()
74      pub_file = os.path.join(keys_dir, f"{algo}_public.key")
75      sec_file = os.path.join(keys_dir, f"{algo}_secret.key")
76      with open(pub_file, "wb") as f:
77          f.write(pk)
78      with open(sec_file, "wb") as f:
79          f.write(sk)
80
81      pk_sha3 = hashlib.sha3_256(pk).digest()
82      _write_rot_c(algo, pk_sha3)
83
84  def do_keygen(args):
85      algo = args.algo.lower()
86      keys_dir = args.keys_dir
87      os.makedirs(keys_dir, exist_ok=True)
88      if algo == "all":
89          print(f"[>] generating all keys into '{keys_dir}'...")
90          for a in ALGO_IDS:
91              _do_keygen_logic(a, keys_dir=keys_dir)
92          print("[>] keys generated.")
93      else:
94          if algo not in ALGO_IDS:
95              raise ValueError(f"unsupported algorithm: {algo}")
96          _do_keygen_logic(algo, keys_dir=keys_dir)
97          print(f"[>] keys for {algo} generated into '{keys_dir}'.")
98
99
100
101 def find_first_last_defined_byte_address_in_hex_region(ih, start: int
    , end: int):
102     first = None
103     last  = None
104     for seg_start, seg_end in ih.segments(): # list of tuple (star
    segment, end segment)
105         clip_start = max(seg_start, start)
106         clip_end   = min(seg_end,   end)
107         if clip_start >= clip_end:
108             continue   # invalid situation. skip iteration, get next
    segment
109         if first is None:                # once set we dont update it
    again
110             first = clip_start           # first is first defined byte
111         last = clip_end - 1              # every clipped segment
    updates last; final one wins
112     return first, last
113
```

```
114 def merge_hex_regions(inhex, c_base=0x80300000, u_base=0xA0300000,
    size=int(1.5*1024*1024)):
115
116     unified = intelhex.IntelHex()
117     first_c, last_c =
    find_first_last_defined_byte_address_in_hex_region(inhex, c_base,
    c_base + size)
118     first_u, last_u =
    find_first_last_defined_byte_address_in_hex_region(inhex, u_base,
    u_base + size)
119     for a in inhex.addresses():   # copy blindly cached region into
    uncached region of "unified"
120         if c_base <= a < c_base + size:
121             unified[a + 0x20000000] = inhex[a]
122
123     for a in inhex.addresses():
124         if u_base <= a < u_base + size:
125             if (a - 0x20000000) in inhex.addresses():
126                 if inhex[a] != inhex[a - 0x20000000]:
127                     raise RuntimeError(f"[!!!] (unify_hex_view)
    mismatch cached/uncach in unify_hex_view() function at address: {a
    }. data is {inhex[a]} vs {inhex[a - 0x20000000]}")
128             else:
129                 unified[a] = inhex[a]
130
131     if len(unified.addresses()) == 0:
132         raise RuntimeError("[!!!] (unify_hex_view) no data found in
    target regions")
133
134     start_addr = min(unified.addresses())     # this is the leftmost
    byte
135     end_addr = max(unified.addresses())       # this is the rightmost
    byte
136
137
138     unified.padding = 0xFF
139
140     return unified, start_addr, end_addr, end_addr - start_addr
141
142
143
144 def pad32(data):
145     rem = len(data) % 32
146     if rem != 0:
147         data.extend(b'\x00' * (32 - rem))
148     return bytearray(data)
149
150 def do_sign(args):
151     algo = args.algo.lower()
```

```
152     if algo not in ALGO_IDS:
153         raise ValueError("[!!!] invalid algo")
154 
155     algo_pk_sz, algo_sk_sz, algo_sig_sz = get_algo_sizes(algo) # read
        from lib
156 
157     skey_path = _resolve_key_path(args.skey, getattr(args, "keys_dir"
        , "keys"))
158     pubkey_path = _resolve_key_path(args.pubkey, getattr(args, "
        keys_dir", "keys"))
159 
160     with open(skey_path, 'rb') as f:
161         sk = f.read()
162     with open(pubkey_path, 'rb') as f:
163         pk = f.read()
164 
165     ih = intelhex.IntelHex(args.ihex)
166 
167     # 0x80300000 total size 1.5MB. 8KB reserved headers = 0x2000.
168     cached_base =   0x80302000
169     uncached_base = 0xA0302000
170     max_binapp_sz = MAX_BIN_SIZE - MACRO_HEADER_SIZE    # 1.5mebi - 8
        kiB
171 
172 
173     unifiex_hex, first_byte_address, last_byte_address, distance =
        merge_hex_regions(ih, cached_base, uncached_base, max_binapp_sz)
174     binapp = unifiex_hex.tobinarray(start=uncached_base, end=
        last_byte_address)
175 
176     binapp = pad32(binapp)
177     if len(binapp) > max_binapp_sz:
178         raise ValueError("[!!!] binary max size exceeded")
179 
180     binapp_hash = hashlib.sha3_256(binapp).digest()
181 
182     version = args.version.split('.')
183     # add zeros if version is incomplete (eg "1.0")
184     while len(version) < 3:
185         version.append("0")
186     v_maj, v_min, v_pat = int(version[0]), int(version[1]), int(
        version[2])
187 
188     # hardcoded program counter addresses for 6 cores
189     core_pcs = [
190         0xA0308300, 0xA0308320, 0xA0308340, 0xFFFFFFFF, 0xFFFFFFFF, 0
        xFFFFFFFF
191     ]
192 
```

```
193     # Pack metadata header (exactly 96 bytes)
194     metadata = build_metadata(
195         binapp_len=len(binapp),
196         v_maj=v_maj,
197         v_min=v_min,
198         v_pat=v_pat,
199         algo_id=ALGO_IDS[algo],
200         hash_id=HASH_SHA3_256,
201         algo_pk_sz=algo_pk_sz,
202         algo_sig_sz=algo_sig_sz,
203         binapp_hash=binapp_hash,
204         core_pcs=core_pcs
205     )
206
207     msg_to_sign = metadata + binapp  # operator + is concat also for
    bytes
208
209     sign_fn = getattr(libpyqcsign, f"{algo}_sign")
210     sig = sign_fn(msg_to_sign, sk)
211
212     # verify immediately
213     verify_fn = getattr(libpyqcsign, f"{algo}_verify")
214     if not verify_fn(sig, msg_to_sign, pk):
215         raise RuntimeError("[!!!] signature verification test failed
    ")
216
217     # blocks aligned to 32 bytes: metadata (96), pubkey, signature
218
219     macro_metadata_header_buf = bytearray(MACRO_HEADER_SIZE)
220
221     offset = 0
222     macro_metadata_header_buf[offset:offset+HEADER_SIZE] = metadata
223     offset += HEADER_SIZE
224
225     pk_padded = pad32(bytearray(pk))
226     macro_metadata_header_buf[offset:offset+len(pk_padded)] =
    pk_padded
227     offset += len(pk_padded)
228
229     # pad to algo_sig_sz, then to 32
230     temp_sig = bytearray(sig)  # sig is bytes. bytes in python are
    immutable. bytearray are mut.
231     if len(temp_sig) < algo_sig_sz:
232         temp_sig.extend(b'\x00' * (algo_sig_sz - len(temp_sig)))
233     elif len(temp_sig) > algo_sig_sz:
234         raise ValueError(f"[!!!] sig sz err : expected {algo_sig_sz}
    bytes, got {len(temp_sig)}")
235     sig_padded = pad32(temp_sig)
236
```

```
237     macro_metadata_header_buf[offset:offset+len(sig_padded)] =
    sig_padded
238     offset += len(sig_padded)
239
240     if offset > (MACRO_HEADER_SIZE - HEADER_SIZE):
241         raise ValueError("[!!!] macro header overflow (pk or sig too
    large for 8KB prefix)")
242
243     # copy metadata header at the bottom of the 8KB header
244     macro_metadata_header_buf[MACRO_HEADER_SIZE - HEADER_SIZE :
    MACRO_HEADER_SIZE] = metadata
245
246     final_output = macro_metadata_header_buf + binapp
247
248     out_dir = args.out_bin_dir
249     os.makedirs(out_dir, exist_ok=True)
250
251     keyname_prefix = ""
252     original_name = os.path.splitext(os.path.basename(args.ihex))[0]
253     out_name = f"{original_name}_signed_{keyname_prefix}{algo}.bin"
254     out_path = os.path.join(out_dir, out_name)
255
256     with open(out_path, 'wb') as f:
257         f.write(final_output)
258
259     print(f"[>] successfully generated signed binary: {out_path}")
260     print(f"[>] questo bin va flashato in slot B. per essere eseguito
    deve essere mosso in slot A")
261
262
263 if __name__ == "__main__":
264     import argparse
265     parser = argparse.ArgumentParser(description="keygen and app
    signer")
266     subparsers = parser.add_subparsers(dest="action", required=True)
267
268     pkgen = subparsers.add_parser("keygen")
269     pkgen.add_argument("algo", type=str, help="algorithm name or 'all
    '")
270     pkgen.add_argument("--keys-dir", type=str, default="keys", )
271
272     psign = subparsers.add_parser("sign")
273     psign.add_argument("--algo",         type=str, required=True,    )
274     psign.add_argument("--skey",         type=str, required=True,    )
275     psign.add_argument("--pubkey",       type=str, required=True,    )
276     psign.add_argument("--ihex",         type=str, required=True,    )
277     psign.add_argument("--version",      type=str, default="1.0.0", )
278     psign.add_argument("--out-bin-dir", type=str, default="out",    )
279     psign.add_argument("--keys-dir",     type=str, default="keys",   )
```

```
280
281     args = parser.parse_args()
282     if args.action == "keygen":
283         do_keygen(args)
284     elif args.action == "sign":
285         do_sign(args)
```

B.2 Python C-Extension Wrapper (`pyqcsign.pyd.c`)

C source code of the `pyqcsign` C-extension module for Python.

Listing B.2: Python C-extension wrapper for cryptographic signature algorithms (`pyqcsign.pyd.c`)

```
1  #include <Python.h>
2  #include <stdint.h>
3
4  /*  MLDSA44 MLDSA65 MLDSA87
5      FALCON512 FALCON1024
6      HAWK512 HAWK1024
7      RSA2048PSS RSA3072PSS RSA2048E3PSS RSA3072E3PSS
8      ED25519
9  */
10
11 /* macro per generare py_NAME_keypair  */
12 #define GEN_KEYPAIR_FUN(NAME) \
13 static PyObject* py_##NAME##_keypair(PyObject* self, PyObject* args)
       { \
14     uint8_t pk[NAME##_PUBLICKEYBYTES]; \
15     uint8_t sk[NAME##_SECRETKEYBYTES]; \
16     if (!PyArg_ParseTuple(args, "")) return NULL; \
17     if (NAME##_crypto_sign_keypair(pk, sk) != 0) { \
18         PyErr_SetString(PyExc_RuntimeError, #NAME " keypair
       generation failed"); \
19         return NULL; \
20     } \
21     return Py_BuildValue("y#y#", pk, (Py_ssize_t)NAME##
       _PUBLICKEYBYTES, \
22                                  sk, (Py_ssize_t)NAME##
       _SECRETKEYBYTES); \
23 }
24
25 /* macro per generare py_NAME_sign_signature  */
26 #define GEN_SIGN_SIGNATURE_FUN(NAME) \
27 static PyObject* py_##NAME##_sign_signature(PyObject* self, PyObject*
       args) { \
28     Py_buffer message, sk; \
29     uint8_t sig[NAME##_BYTES]; \
```

```
30      size_t siglen; \
31      if (!PyArg_ParseTuple(args, "y*y*", &message, &sk)) return NULL;
    \
32      if (sk.len != (Py_ssize_t)NAME##_SECRETKEYBYTES) { \
33          PyBuffer_Release(&message); PyBuffer_Release(&sk); \
34          PyErr_SetString(PyExc_ValueError, "invalid secret key length"
    ); \
35          return NULL; \
36      } \
37      if (NAME##_crypto_sign_signature(sig, (void*)&siglen, (uint8_t*)
    message.buf, (size_t)message.len, (uint8_t*)sk.buf) != 0) { \
38          PyBuffer_Release(&message); PyBuffer_Release(&sk); \
39          PyErr_SetString(PyExc_RuntimeError, #NAME " signature failed"
    ); \
40          return NULL; \
41      } \
42      PyBuffer_Release(&message); \
43      PyBuffer_Release(&sk); \
44      return PyBytes_FromStringAndSize((char*)sig, (Py_ssize_t)siglen);
        \
45  }
46
47  /* macro per generare py_NAME_verify  */
48  #define GEN_VERIFY_FUN(NAME) \
49  static PyObject* py_##NAME##_verify(PyObject* self, PyObject* args) {
        \
50      Py_buffer sig, message, pk; \
51      if (!PyArg_ParseTuple(args, "y*y*y*", &sig, &message, &pk))
    return NULL; \
52      if (pk.len != (Py_ssize_t)NAME##_PUBLICKEYBYTES) { \
53          PyBuffer_Release(&sig); PyBuffer_Release(&message);
    PyBuffer_Release(&pk); \
54          PyErr_SetString(PyExc_ValueError, "invalid public key length"
    ); \
55          return NULL; \
56      } \
57      int result = NAME##_crypto_sign_verify((uint8_t*)sig.buf, (size_t
    )sig.len, (uint8_t*)message.buf, (size_t)message.len, (uint8_t*)pk
    .buf); \
58      PyBuffer_Release(&sig); PyBuffer_Release(&message);
    PyBuffer_Release(&pk); \
59      if (result == 0) Py_RETURN_TRUE; \
60      else Py_RETURN_FALSE; \
61  }
62
63
64  /* esempio  per MLDSA44: */
65  GEN_KEYPAIR_FUN(MLDSA44)
66  GEN_SIGN_SIGNATURE_FUN(MLDSA44)
```

```
67 GEN_VERIFY_FUN(MLDSA44)
68
69
70 /* macro per generare le voci di PyMethodDef */
71 #define GEN_METHOD_ENTRIES(NAME, name) \
72     {#name "_keypair", py_##NAME##_keypair,          METH_VARARGS, "
       generate " #name " keypair."}, \
73     {#name "_sign",    py_##NAME##_sign_signature, METH_VARARGS, "
       sign message using " #name "."}, \
74     {#name "_verify",  py_##NAME##_verify,          METH_VARARGS, "
       verify " #name " signature."}
75
76 /* macro per registrare le costanti di ciascun algoritmo */
77 #define ADD_INT_CONSTANTS(NAME, CONST_PREFIX) do { \
78     PyModule_AddIntConstant(module, #NAME "_PUBLICKEYBYTES",
       CONST_PREFIX##_PUBLICKEYBYTES); \
79     PyModule_AddIntConstant(module, #NAME "_SECRETKEYBYTES",
       CONST_PREFIX##_SECRETKEYBYTES); \
80     PyModule_AddIntConstant(module, #NAME "_BYTES",
       CONST_PREFIX##_BYTES); \
81 } while(0)
82
83
84 static PyMethodDef module_method_list[] = {
85     GEN_METHOD_ENTRIES(MLDSA44, mldsa44),
86     /*GEN_METHOD_ENTRIES(MLDSA65, mldsa65),*/
87     /*GEN_METHOD_ENTRIES(MLDSA87, mldsa87),*/
88     /*...*/
89     {NULL, NULL, 0, NULL}
90 };
91
92 static struct PyModuleDef module_struct = {
93     PyModuleDef_HEAD_INIT,
94     "pyqcsign",
95     "python function wrapper c lang pqc and not-pqc sign algo",
96     -1,
97     module_method_list
98 };
99
100 PyMODINIT_FUNC PyInit_pyqcsign(void) {
101     PyObject *module;
102
103     module = PyModule_Create(&module_struct);
104     if (module == NULL) return NULL;
105
106     /* constants added to the module so they are available in python
       script */
107     ADD_INT_CONSTANTS(MLDSA44, PQCLEAN_MLDSA44_CLEAN_CRYPTO);
108     /*ADD_INT_CONSTANTS(MLDSA65, PQCLEAN_MLDSA65_CLEAN_CRYPTO);*/
```

```
109         /*ADD_INT_CONSTANTS(MLDSA87, PQCLEAN_MLDSA87_CLEAN_CRYPTO);*/
110         /* ... */
111
112         return module;
113 }
```

Appendix C

AURIX Benchmark Harness

C.1 CPU Performance Counter Harness

Listing C.1: AURIX CPU cycle measurement harness macro (`Perf_Counter_Harness.h`)

```
1   /* usage:  ACCU_MEASURE( accumulator[idx],
2    *                [...code goes here...]
3    *            );
4    */
5  #define ACCU_MEASURE(OUTPUT, ...)\
6      do{\
7          IfxCpu_Perf perf_tmp; \
8          IfxCpu_resetAndStartCounters(IfxCpu_CounterMode_normal); \
9          __VA_ARGS__ \
10         perf_tmp = IfxCpu_stopCounters(); \
11         if(perf_tmp.clock.overlfow==0) OUTPUT += perf_tmp.clock.
       counter; \
12         else OUTPUT=0; \
13     }while(0)
```

Appendix D

AURIX Assembly NTT Implementation

D.1 Forward NTT Assembly Routine

Forward Number Theoretic Transform (NTT) for ML-DSA.

Listing D.1: AURIX TriCore assembly implementation of the forward NTT (`mldsa_asm_ntt_.src`)

```
.global asm_ntt

.sdecl ".text", CODE
.sect ".text"
.align 4

asm_ntt:.type func
    svlcx
    .include 'mldsa_asm_ntt_utils.src'

    .align 4

    .define pol_arr 'a4'
    .define zeta_arr 'a5'

    .define pol0 'd8'
    .define pol1 'd9'
    .define pol2 'd10'
    .define pol3 'd11'
    .define pol4 'd12'
    .define pol5 'd13'
    .define pol6 'd14'
    .define pol7 'd15'
```

```
24
25      .define zeta0 'a2'
26      .define zeta1 'a3'
27      .define zeta2 'a11'
28      .define zeta3 'a6'
29      .define zeta4 'a7'
30      .define zeta5 'a12'
31      .define zeta6 'a13'     .define temp_addr 'a14'
32      .define temp_addr_2 'a15'
33
34      .define Qinv 'd5'
35      .define Qq 'd4'
36      distance .SET 512
37      strincr .SET 4
38      ld.a zeta0 , [zeta_arr+](1*4)
39      ld.a zeta1 , [zeta_arr+](1*4)
40      ld.a zeta2 , [zeta_arr+](1*4)
41      ld.a zeta3 , [zeta_arr+](1*4)
42      ld.a zeta4 , [zeta_arr+](1*4)
43      ld.a zeta5 , [zeta_arr+](1*4)
44      ld.a zeta6 , [zeta_arr+](1*4)
45
46      mov scratch1 , #(32*strincr)
47      ADDSC.A temp_addr, pol_arr, scratch1 , #0
48
49      stage1:
50          .DUP 2
51          ld.w pol0 , [pol_arr](0*distance/4)
52          ld.w pol1 , [pol_arr](1*distance/4)
53          ld.w pol2 , [pol_arr](2*distance/4)
54          ld.w pol3 , [pol_arr](3*distance/4)
55          ld.w pol4 , [pol_arr](4*distance/4)
56          ld.w pol5 , [pol_arr](5*distance/4)
57          ld.w pol6 , [pol_arr](6*distance/4)
58          ld.w pol7 , [pol_arr](7*distance/4)
59
60          _3_layer_CT_32 pol0 ,pol1 ,pol2 ,pol3 ,pol4 ,pol5 ,pol6 ,pol7 ,zeta0 ,
        zeta1 ,zeta2 ,zeta3 ,zeta4 ,zeta5 ,zeta6 ,Qinv ,Qq
61
62          st.w [pol_arr](0*distance/4), pol0
63          st.w [pol_arr](1*distance/4), pol1
64          st.w [pol_arr](2*distance/4), pol2
65          st.w [pol_arr](3*distance/4), pol3
66          st.w [pol_arr](4*distance/4), pol4
67          st.w [pol_arr](5*distance/4), pol5
68          st.w [pol_arr](6*distance/4), pol6
69          st.w [pol_arr](7*distance/4), pol7
70
71          add.a pol_arr , #(strincr)          .ENDM
```

```
72
73      jne.a pol_arr, temp_addr, stage1      movh.a temp_addr,#@HI(32*4)
74      lea temp_addr, [temp_addr]@LO(32*4)
75      sub.a pol_arr,pol_arr,temp_addr      distance .SET 64      mov
        scratch1, #(8*112+8*4*4)
76      ADDSC.A temp_addr, pol_arr, scratch1, #0
77
78      stage4_1:
79          mov scratch1, #(4*strincr)
80          ADDSC.A temp_addr_2, pol_arr, scratch1, #0
81          ld.a zeta0 , [zeta_arr+](1*4)
82          ld.a zeta1 , [zeta_arr+](1*4)
83          ld.a zeta2 , [zeta_arr+](1*4)
84          ld.a zeta3 , [zeta_arr+](1*4)
85          ld.a zeta4 , [zeta_arr+](1*4)
86          ld.a zeta5 , [zeta_arr+](1*4)
87          ld.a zeta6 , [zeta_arr+](1*4)          stage4_2:
88              .DUP 2
89              ld.w pol0, [pol_arr](0*distance/4)
90              ld.w pol1, [pol_arr](1*distance/4)
91              ld.w pol2, [pol_arr](2*distance/4)
92              ld.w pol3, [pol_arr](3*distance/4)
93              ld.w pol4, [pol_arr](4*distance/4)
94              ld.w pol5, [pol_arr](5*distance/4)
95              ld.w pol6, [pol_arr](6*distance/4)
96              ld.w pol7, [pol_arr](7*distance/4)
        _3_layer_CT_32 pol0,pol1,pol2,pol3,pol4,pol5,pol6,pol7,zeta0,zeta1
        ,zeta2,zeta3,zeta4,zeta5,zeta6,Qinv,Qq
97
98              st.w [pol_arr](0*distance/4), pol0
99              st.w [pol_arr](1*distance/4), pol1
100             st.w [pol_arr](2*distance/4), pol2
101             st.w [pol_arr](3*distance/4), pol3
102             st.w [pol_arr](4*distance/4), pol4
103             st.w [pol_arr](5*distance/4), pol5
104             st.w [pol_arr](6*distance/4), pol6
105             st.w [pol_arr](7*distance/4), pol7
106
107             mov scratch1, #4
108             ADDSC.A pol_arr, pol_arr, scratch1, #0
109
110             .ENDM
111
112         jne.a pol_arr, temp_addr_2, stage4_2
113
114         mov scratch1, #112
115         ADDSC.A pol_arr,pol_arr, scratch1, #0
116     jne.a pol_arr, temp_addr, stage4_1      movh.a temp_addr, #@HI
        (4*4*8+112*8)
```

```
117     lea temp_addr, [temp_addr]@LO(4*4*8+112*8)
118     sub.a pol_arr, pol_arr, temp_addr     mov scratch1, #(1024)
119     ADDSC.A temp_addr, pol_arr, scratch1, #0
120
121     stage7:
122
123         ld.a zeta1, [zeta_arr]4
124         ld.a zeta2, [zeta_arr]8
125         ld.a zeta0, [zeta_arr]0
126         mov scratch1, #12
127         addsc.a zeta_arr, zeta_arr, scratch1, #0
128
129         ld.w pol0, [pol_arr]0
130         ld.w pol1, [pol_arr]4
131         ld.w pol2, [pol_arr]8
132         ld.w pol3, [pol_arr]12
133
134         _2_layer_CT_32 pol0,pol1,pol2,pol3,zeta0,zeta1,zeta2,Qinv,Qq
135
136         st.w  [pol_arr]4, pol1
137         st.w  [pol_arr]8, pol2
138         st.w  [pol_arr]12, pol3
139         st.w  [pol_arr]0, pol0
140
141         mov scratch1, #16
142         ADDSC.A pol_arr,pol_arr, scratch1, #0
143
144     jne.a temp_addr, pol_arr, stage7
145
146
147     rslcx
148     ret
```

D.2 Inverse NTT Assembly Routine

Inverse Number Theoretic Transform (iNTT) for ML-DSA.

Listing D.2: AURIX TriCore assembly implementation of the inverse NTT (`mldsa_asm_ntt_inv.src`)

```
1 .global asm_invntt
2
3 .sdecl ".text", CODE
4 .sect ".text"
5 .align 4
6
7 asm_invntt:.type func
8     svlcx
```

```
9       .include 'mldsa_asm_ntt_utils.src'
10
11      .define pol_arr 'a4'
12      .define zeta_arr 'a5'
13
14      .define pol0 'd8'
15      .define pol1 'd9'
16      .define pol2 'd10'
17      .define pol3 'd11'
18      .define pol4 'd12'
19      .define pol5 'd13'
20      .define pol6 'd14'
21      .define pol7 'd15'
22
23      .define zeta0 'a2'
24      .define zeta1 'a3'
25      .define zeta2 'a11'
26      .define zeta3 'a6'
27      .define zeta4 'a7'
28      .define zeta5 'a12'
29      .define zeta6 'a13'
30      .define zeta7 'a14'
31
32      .define temp_addr 'a14'
33      .define temp_addr_2 'a15'
34
35      .define Qinv 'd5'
36      .define Qq 'd4'
37      distance .SET 16
38      strincr .SET 32
39      ld.a zeta0 , [zeta_arr+](1*4)
40      ld.a zeta1 , [zeta_arr+](1*4)
41      ld.a zeta2 , [zeta_arr+](1*4)
42      ld.a zeta3 , [zeta_arr+](1*4)
43      ld.a zeta4 , [zeta_arr+](1*4)
44      ld.a zeta5 , [zeta_arr+](1*4)
45      ld.a zeta6 , [zeta_arr+](1*4)
46      mov scratch1, #(32*strincr)
47      addsc.a temp_addr, pol_arr, scratch1, #0
48
49      stage1:
50          ld.w pol4, [pol_arr](4*distance/4)
51          ld.w pol1, [pol_arr](5*distance/4)
52          ld.w pol6, [pol_arr](6*distance/4)
53          ld.w pol3, [pol_arr](7*distance/4)
54          _3_layer_inv_butterfly_light_fast_first pol0, pol1, pol2,
        pol3, pol4, pol5, pol6, pol7,zeta0,zeta1,zeta2,zeta3,zeta4,zeta5,
        zeta6,Qinv,Qq
55
```

```
56          ld.w  pol0 , [pol_arr](0*distance/4)
57          ld.w  pol1 , [pol_arr](1*distance/4)
58          ld.w  pol2 , [pol_arr](2*distance/4)
59          ld.w  pol3 , [pol_arr](3*distance/4)
60          _3_layer_inv_butterfly_light_fast_second  pol0 , pol1 , pol2 ,
    pol3 , pol4 , pol5 , pol6 , pol7 , zeta0 , zeta1 , zeta2 , zeta3 , zeta4 , zeta5 ,
    zeta6 , Qinv , Qq
61
62          st.w  [pol_arr](0*distance/4), pol0
63          st.w  [pol_arr](1*distance/4), pol1
64          st.w  [pol_arr](2*distance/4), pol2
65          st.w  [pol_arr](3*distance/4), pol3
66          st.w  [pol_arr](4*distance/4), pol4
67          st.w  [pol_arr](5*distance/4), pol5
68          st.w  [pol_arr](6*distance/4), pol6
69          st.w  [pol_arr](7*distance/4), pol7
70          mov scratch1 , #strincr
71          addsc.a  pol_arr , pol_arr , scratch1 , #0
72
73      jne.a pol_arr , temp_addr , stage1       movh.a temp_addr,#@HI(32*
    strincr)
74      lea temp_addr , [temp_addr]@LO(32*strincr)
75      sub.a pol_arr , pol_arr , temp_addr
76      distance .SET 128
77      strincr  .SET 256     mov scratch1 , #(4*256)
78      addsc.a temp_addr , pol_arr , scratch1 , #0
79      ld.a zeta0 , [zeta_arr+](1*4)
80      ld.a zeta1 , [zeta_arr+](1*4)
81      ld.a zeta2 , [zeta_arr+](1*4)
82      ld.a zeta3 , [zeta_arr+](1*4)
83      ld.a zeta4 , [zeta_arr+](1*4)
84      ld.a zeta5 , [zeta_arr+](1*4)
85      ld.a zeta6 , [zeta_arr+](1*4)
86
87      stage4_2:
88          ld.w  pol4 ,  [pol_arr](4*distance/4)
89          ld.w  pol1 ,  [pol_arr](5*distance/4)
90          ld.w  pol6 ,  [pol_arr](6*distance/4)
91          ld.w  pol3 ,  [pol_arr](7*distance/4)
92          _3_layer_inv_butterfly_light_fast_first  pol0 , pol1 , pol2 ,
    pol3 , pol4 , pol5 , pol6 , pol7 , zeta0 , zeta1 , zeta2 , zeta3 , zeta4 , zeta5 ,
    zeta6 , Qinv , Qq
93
94          ld.w  pol0 ,  [pol_arr](0*distance/4)
95          ld.w  pol1 ,  [pol_arr](1*distance/4)
96          ld.w  pol2 ,  [pol_arr](2*distance/4)
97          ld.w  pol3 ,  [pol_arr](3*distance/4)
```

```
98          _3_layer_inv_butterfly_light_fast_second pol0, pol1, pol2,
    pol3, pol4, pol5, pol6, pol7,zeta0,zeta1,zeta2,zeta3,zeta4,zeta5,
    zeta6,Qinv,Qq
99
100         st.w [pol_arr](0*distance/4), pol0
101         st.w [pol_arr](1*distance/4), pol1
102         st.w [pol_arr](2*distance/4), pol2
103         st.w [pol_arr](3*distance/4), pol3
104         st.w [pol_arr](4*distance/4), pol4
105         st.w [pol_arr](5*distance/4), pol5
106         st.w [pol_arr](6*distance/4), pol6
107         st.w [pol_arr](7*distance/4), pol7
108        mov scratch1, #(strincr)
109         addsc.a pol_arr, pol_arr, scratch1, #0
110
111     jne.a pol_arr, temp_addr, stage4_2      movh.a temp_addr,#@HI
    (4*256-4)
112     lea temp_addr, [temp_addr]@LO(4*256-4)
113     sub.a pol_arr, pol_arr, temp_addr      mov scratch1, #(7*4)
114     addsc.a temp_addr, pol_arr, scratch1, #0
115
116     stage4_it1_1:          mov scratch1, #(4*strincr)
117         addsc.a temp_addr_2, pol_arr, scratch1, #0
118
119         ld.a zeta0 , [zeta_arr+](1*4)
120         ld.a zeta1 , [zeta_arr+](1*4)
121         ld.a zeta2 , [zeta_arr+](1*4)
122         ld.a zeta3 , [zeta_arr+](1*4)
123         ld.a zeta4 , [zeta_arr+](1*4)
124         ld.a zeta5 , [zeta_arr+](1*4)
125         ld.a zeta6 , [zeta_arr+](1*4)
126
127         stage4_it1_2:
128
129             ld.w pol0, [pol_arr](0*distance/4)
130             ld.w pol1, [pol_arr](1*distance/4)
131             ld.w pol2, [pol_arr](2*distance/4)
132             ld.w pol3, [pol_arr](3*distance/4)
133             ld.w pol4, [pol_arr](4*distance/4)
134             ld.w pol5, [pol_arr](5*distance/4)
135             ld.w pol6, [pol_arr](6*distance/4)
136             ld.w pol7, [pol_arr](7*distance/4)
137
138             _3_layer_inv_CT_32 pol0, pol1, pol2, pol3, pol4, pol5,
    pol6, pol7,zeta0,zeta1,zeta2,zeta3,zeta4,zeta5,zeta6,Qinv,Qq
139
140             st.w [pol_arr](0*distance/4), pol0
141             st.w [pol_arr](1*distance/4), pol1
142             st.w [pol_arr](2*distance/4), pol2
```

```
143             st.w [pol_arr](3*distance/4), pol3
144             st.w [pol_arr](4*distance/4), pol4
145             st.w [pol_arr](5*distance/4), pol5
146             st.w [pol_arr](6*distance/4), pol6
147             st.w [pol_arr](7*distance/4), pol7               mov
    scratch1, #strincr
148             ADDSC.A pol_arr,pol_arr, scratch1, #0
149         jne.a pol_arr, temp_addr_2, stage4_it1_2
150
151         movh.a temp_addr_2, #@HI(4*strincr-4)
152         lea temp_addr_2, [temp_addr_2]@LO(4*strincr-4)
153         sub.a pol_arr, pol_arr, temp_addr_2
154
155     jne.a pol_arr, temp_addr, stage4_it1_1
156
157     movh.a temp_addr, #@HI(8*4)
158     lea  temp_addr, [temp_addr]@LO(8*4)
159     sub.a pol_arr, pol_arr, temp_addr
160     strincr .SET 4     mov scratch1, #(64*strincr)
161     ADDSC.A temp_addr, pol_arr, scratch1, #0
162
163     stage7:
164
165         ld.a zeta1, [zeta_arr]4
166         ld.a zeta2, [zeta_arr]8
167         ld.a zeta0, [zeta_arr]
168         mov scratch1, #12
169         addsc.a zeta_arr, zeta_arr, scratch1, #0
170
171         ld.w pol0, [pol_arr]0
172         ld.w pol1, [pol_arr]256
173         ld.w pol2, [pol_arr]512
174         ld.w pol3, [pol_arr]768
175
176         _2_layer_inv_CT_32 pol0, pol1, pol2, pol3, zeta0, zeta1,
    zeta2, Qinv, Qq
177         ld.w scratch1, [zeta_arr]0
178         montgomery_mul_32 pol0, scratch1   , Qinv, Qq
179
180         ld.w scratch1, [zeta_arr]4
181         montgomery_mul_32 pol1, scratch1         , Qinv,Qq
182
183         ld.w scratch1, [zeta_arr]8
184         montgomery_mul_32 pol2, scratch1         , Qinv,Qq
185
186         ld.w scratch1, [zeta_arr]12
187         montgomery_mul_32 pol3, scratch1         , Qinv,Qq
188
189         mov scratch1, #16
```

```
190         addsc.a zeta_arr, zeta_arr, scratch1, #0
191
192         st.w [pol_arr]256, pol1
193         st.w [pol_arr]512, pol2
194         st.w [pol_arr]768, pol3
195         st.w [pol_arr]0   , pol0
196         mov scratch1, #(strincr)
197         addsc.a pol_arr, pol_arr, scratch1, #0
198
199
200     jne.a pol_arr, temp_addr, stage7
201
202     rslcx
203     ret
```

D.3 NTT Assembly Utilities and Macros

Montgomery multiplication, butterfly operations, and CT / GS layers used in the ML-DSA NTT implementations on AURIX TriCore.

Listing D.3: AURIX TriCore assembly macros for ML-DSA NTT (`mldsa_asm_ntt_utils.src`)

```
1  .define A 'd1'
2  .define t 'd0'
3  .define t2 'd2'
4  .define scratch1 'd6'
5  .define scratch2 'd7'
6  .define twiddle 'd3'
7  montgomery_mul_32 .macro  a, b, qp, q
8
9      mul E0,\a,\b
10     mul t2,t,\qp
11     madd E0,E0,\t2,\q
12     mov \a,\A
13 .endm
14
15 addSub1 .macro  c0, c1
16     add \c0, \c1
17     sh \c1,#1
18     sub \c1, \c0, \c1
19 .endm
20
21 addSub2 .macro  c0, c1, c2, c3
22     add \c0, \c1
23     add \c2, \c3
24     sh \c1, #1
```

```
25      sh \c3, #1
26      sub \c1, \c0, \c1
27      sub \c3, \c2, \c3
28
29  .endmaddSub4 .macro c0, c1, c2, c3, c4, c5, c6, c7
30      add \c0, \c1
31      add \c2, \c3
32      add \c4, \c5
33      add \c6, \c7
34      sh \c1, #1
35      sh \c3, #1
36      sh \c5, #1
37      sh \c7, #1
38      sub \c1, \c0, \c1
39      sub \c3, \c2, \c3
40      sub \c5, \c4, \c5
41      sub \c7, \c6, \c7
42  .endm
43
44  _2_layer_CT_32 .macro c0, c1, c2, c3, zeta0, zeta1, zeta2, Qprime,
        Qq
45      mov.d twiddle,\zeta0
46      montgomery_mul_32 \c2, twiddle, \Qprime, \Qq
47      mov.d twiddle, \zeta0
48      montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
49      addSub2 \c0, \c2, \c1, \c3
50
51      mov.d twiddle, \zeta1
52      montgomery_mul_32 \c1, twiddle, \Qprime, \Qq
53      mov.d twiddle, \zeta2
54      montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
55      addSub2 \c0, \c1, \c2, \c3
56  .endm
57
58  _2_layer_inv_CT_32 .macro c0, c1, c2, c3, zeta0, zeta1, zeta2, Qprime
        , Qq
59      mov.d twiddle, zeta0
60      montgomery_mul_32 \c1, twiddle, \Qprime, \Qq
61      montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
62      addSub2 \c0, \c1, \c2, \c3
63
64      mov.d twiddle, zeta1
65      montgomery_mul_32 \c2, twiddle, \Qprime, \Qq
66      mov.d twiddle, zeta2
67      montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
68      addSub2 \c0, \c2, \c1, \c3
69  .endm
70
```

```
71 _3_layer_CT_32 .macro c0, c1, c2, c3, c4, c5, c6, c7, z0, z1, z2, z3,
       z4, z5, z6, Qprime, Qq
72     mov.d twiddle, \z0
73     montgomery_mul_32 \c4, twiddle, \Qprime, \Qq
74     montgomery_mul_32 \c5, twiddle, \Qprime, \Qq
75     montgomery_mul_32 \c6, twiddle, \Qprime, \Qq
76     montgomery_mul_32 \c7, twiddle, \Qprime, \Qq
77     addSub4 \c0, \c4, \c1, \c5, \c2, \c6, \c3, \c7
78
79     mov.d twiddle, \z1
80     montgomery_mul_32 \c2, twiddle, \Qprime, \Qq
81     montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
82     mov.d twiddle, \z2
83     montgomery_mul_32 \c6, twiddle, \Qprime, \Qq
84     montgomery_mul_32 \c7, twiddle, \Qprime, \Qq
85     addSub4 \c0, \c2, \c1, \c3, \c4, \c6, \c5, \c7
86
87     mov.d twiddle, \z3
88     montgomery_mul_32 \c1, twiddle, \Qprime, \Qq
89     mov.d twiddle, \z4
90     montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
91     mov.d twiddle, \z5
92     montgomery_mul_32 \c5, twiddle, \Qprime, \Qq
93     mov.d twiddle, \z6
94     montgomery_mul_32 \c7, twiddle, \Qprime, \Qq
95     addSub4 \c0, \c1, \c2, \c3, \c4, \c5, \c6, \c7
96
97 .endm
98
99 _3_layer_inv_CT_32 .macro c0, c1, c2, c3, c4, c5, c6, c7, zeta0,
       zeta1, zeta2, zeta3, zeta4, zeta5, zeta6, Qprime, Qq
100    mov.d twiddle, zeta0
101    montgomery_mul_32 \c1, twiddle, \Qprime, \Qq
102    montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
103    montgomery_mul_32 \c5, twiddle, \Qprime, \Qq
104    montgomery_mul_32 \c7, twiddle, \Qprime, \Qq
105    addSub4 \c0, \c1, \c2, \c3, \c4, \c5, \c6, \c7
106
107    mov.d twiddle, \zeta1
108    montgomery_mul_32 \c2, twiddle, \Qprime, \Qq
109    montgomery_mul_32 \c6, twiddle, \Qprime, \Qq
110    mov.d twiddle, \zeta2
111    montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
112    montgomery_mul_32 \c7, twiddle, \Qprime, \Qq
113    addSub4 \c0, \c2, \c1, \c3, \c4, \c6, \c5, \c7
114
115    mov.d twiddle, \zeta3
116    montgomery_mul_32 \c4, twiddle, \Qprime, \Qq
117    mov.d twiddle, \zeta4
```

```
118     montgomery_mul_32 \c5, twiddle, \Qprime, \Qq
119     mov.d twiddle, \zeta5
120     montgomery_mul_32 \c6, twiddle, \Qprime, \Qq
121     mov.d twiddle, \zeta6
122     montgomery_mul_32 \c7, twiddle, \Qprime, \Qq
123     addSub4 \c0, \c4, \c1, \c5, \c2, \c6, \c3, \c7
124
125 .endm_3_layer_inv_butterfly_light_fast_first .macro c0, c1, c2, c3,
        c4, c5, c6, c7, z0, z1, z2, z3, z4, z5, z6, Qprime, Qq
126     addSub2 \c4, \c1, \c6, \c3
127     addSub1 \c4, \c6
128
129     mov.d scratch1, z4
130     mov.d scratch2, z6
131
132     mul  E0,     \c1, scratch1
133     madd E0, E0, \c3, scratch2
134     mul twiddle, d0, \Qprime
135     madd E0, E0, twiddle, \Qq
136     mov \c0, d0
137     mov \c5, d1
138
139     mul  E0,    c1, scratch2
140     madd E0,E0,\c3, scratch1
141     mul twiddle, d0, \Qprime
142     madd E0, E0, twiddle, \Qq
143     mov \c2, d0
144     mov \c7, d1
145
146 .endm_3_layer_inv_butterfly_light_fast_second .macro c0, c1, c2, c3,
        c4, c5, c6, c7, z0, z1, z2, z3, z4, z5, z6, Qprime, Qq
147     addSub2 \c0, \c1, \c2, \c3
148
149     mov.d twiddle, \zeta2
150     montgomery_mul_32 \c3, twiddle, \Qprime, \Qq
151     addSub2 \c0, \c2, \c1, \c3
152
153     montgomery_mul_32 \c6, twiddle, \Qprime, \Qq
154
155     addSub4 \c0, \c4, \c1, \c5, \c2, \c6, \c3, \c7
156 .endm
```

Bibliography

[1] Steffen Sanwald, Liron Kaneti, Marc Stöttinger, and Martin Böhner. «Secure Boot Revisited: Challenges for Secure Implementations in the Automotive Domain». English. In: *SAE International Journal of Transportation Cybersecurity and Privacy* 2.2 (Aug. 2020). Number: 11-02-02-0008, pp. 69–81. ISSN: 2572-1046, 2572-1054. DOI: 10.4271/11-02-02-0008. URL: https://www.sae.org/publications/technical-papers/content/11-02-02-0008/ (visited on 01/07/2025) (cit. on pp. 1, 4, 42, 46, 51, 95).

[2] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. *A Firmware Update Architecture for Internet of Things*. RFC 9019. Apr. 2021. DOI: 10.17487/RFC9019. URL: https://www.rfc-editor.org/info/rfc9019 (cit. on pp. 1, 38–41, 46).

[3] Kai Tang, Xin Xu, and Chunxia Guo. «The Secure Boot of Embedded System Based on Mobile Trusted Module». In: *2012 Second International Conference on Intelligent System Design and Engineering Application* (Jan. 2012). Conference Name: 2012 Second International Conference on Intelligent System Design and Engineering Application (ISDEA) ISBN: 9781457721205 9780769546087 Place: Sanya, China Publisher: IEEE, pp. 1331–1334. DOI: 10.1109/ISdea.2012.646. URL: http://ieeexplore.ieee.org/document/6173454/ (visited on 01/07/2025) (cit. on p. 1).

[4] United Nations Economic Commission for Europe. *UN Regulation No. 155 – Uniform Provisions Concerning the Approval of Vehicles with Regard to Cyber Security and Cyber Security Management System*. Tech. rep. E/ECE/TRANS/505/Rev.3/Ad UNECE World Forum for Harmonization of Vehicle Regulations (WP.29), 2021. URL: https://unece.org/transport/documents/2021/03/standards/un-regulation-no-155-cyber-security-and-cyber-security (cit. on p. 2).

[5] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on Post-Quantum Cryptography*. Tech. rep. NISTIR 8105. National Institute of Standards and Technology,

2016. DOI: 10.6028/NIST.IR.8105. URL: https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf (cit. on pp. 2, 7, 8).

[6] Peter W. Shor. «Algorithms for quantum computation: discrete logarithms and factoring». In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. IEEE, 1994, pp. 124–134 (cit. on pp. 2, 7, 8).

[7] Michele Mosca. «Cybersecurity in an Era with Quantum Computers: Will We Be Ready?» In: *IEEE Security & Privacy* 16.5 (2018). Free preprint: IACR ePrint 2015/1075, https://eprint.iacr.org/2015/1075, pp. 38–41. DOI: 10.1109/MSP.2018.3761723 (cit. on p. 2).

[8] Michele Mosca and Marco Piani. *2024 Quantum Threat Timeline Report*. Tech. rep. Global Risk Institute, 2024. URL: https://globalriskinstitute.org/publication/2024-quantum-threat-timeline-report/ (cit. on pp. 2, 94).

[9] *AURIX TC3xx 32-Bit Microcontroller Family User's Manual*. Version 1.6.0. Infineon Technologies AG. 2020. URL: https://www.infineon.com/ (cit. on pp. 2, 47, 50).

[10] *HSM – Hardware Security Module – AURIX™ TC3xx Microcontroller Training*. Version 1.0. Infineon Technologies AG. Sept. 2020. URL: https://www.infineon.com/dgdl/Infineon-HSM-Hardware_Security_Module-Training-v01_00-EN.pdf?fileId=5546d462743a6d7101746b142517173e (cit. on p. 2).

[11] Infineon Technologies AG. *Infineon Automotive Security Compendium: Protective Measures Against Cyberattacks*. [Online; whitepaper version 01.00]. Aug. 2020. URL: https://www.allaboutcircuits.com/uploads/articles/Infineon-ATV_Cyber_Security_Compendium_Public-Whitepaper-v01_00-EN.pdf (cit. on pp. 2, 58, 59, 99).

[12] Vinay B. Y. Kumar, Naina Gupta, Anupam Chattopadhyay, Michael Kasper, Christoph Krauß, and Ruben Niederhagen. «Post-Quantum Secure Boot». In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. ISSN: 1558-1101. Mar. 2020, pp. 1582–1585. DOI: 10.23919/DATE48585.2020.9116252. URL: https://ieeexplore.ieee.org/document/9116252 (visited on 01/07/2025) (cit. on pp. 2, 4, 95).

[13] Kirsten Matheus and Thomas Königseder. *Automotive Ethernet*. 2nd ed. Second edition. Cambridge University Press, 2017. ISBN: 9781107148321 (cit. on p. 2).

[14] Dennis Kengo Oka and Philipp Jungklass. «Secure Boot in Automotive Real-Time Systems in Context of Post-Quantum Cryptography». In: *Proceedings of the 2nd Cyber Security in CarS Workshop (CSCS '25)*. Association for Computing Machinery, 2025. DOI: 10.1145/3736130.3764513 (cit. on pp. 2, 5, 91, 92, 95).

[15] Soundes Marzougui and Juliane Krämer. «Post-Quantum Cryptography in Embedded Systems». In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. ARES '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 1–7. ISBN: 978-1-4503-7164-3. DOI: 10.1145/3339252.3341475. URL: https://doi.org/10.1145/3339252.3341475 (visited on 01/07/2025) (cit. on pp. 3, 5, 96).

[16] David A. Cooper, Daniel C. Apon, Quynh H. Dang, Michael S. Davidson, Morris J. Dworkin, and Carl A. Miller. *Recommendation for Stateful Hash-Based Signature Schemes*. Tech. rep. NIST Special Publication 800-208. National Institute of Standards and Technology, Oct. 2020. DOI: 10.6028/NIST.SP.800-208. URL: https://doi.org/10.6028/NIST.SP.800-208 (cit. on pp. 3, 99).

[17] National Security Agency. *Announcing the Commercial National Security Algorithm Suite 2.0*. Cybersecurity Advisory. National Security Agency, Sept. 2022. URL: https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS.PDF (cit. on pp. 3, 99).

[18] National Institute of Standards and Technology. *Stateless Hash-Based Digital Signature Standard*. Tech. rep. FIPS 205. National Institute of Standards and Technology, Aug. 2024. DOI: 10.6028/NIST.FIPS.205. URL: https://doi.org/10.6028/NIST.FIPS.205 (cit. on pp. 4, 10).

[19] Jessie Frazelle. «Securing the boot process». In: *Commun. ACM* 63.3 (Feb. 2020), pp. 38–42. ISSN: 0001-0782. DOI: 10.1145/3379512. URL: https://dl.acm.org/doi/10.1145/3379512 (visited on 01/07/2025) (cit. on p. 4).

[20] Tim Fritzmann, Jonas Vith, Daniel Flórez, and Johanna Sepúlveda. «Post-quantum cryptography for automotive systems». In: *Microprocessors and Microsystems* 87 (Nov. 2021), p. 104379. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2021.104379. URL: https://www.sciencedirect.com/science/article/pii/S0141933121005299 (visited on 01/07/2025) (cit. on pp. 4, 91).

[21] Tim Fritzmann, Jonas Vith, and Johanna Sepúlveda. «Strengthening Post-Quantum Security for Automotive Systems». In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. Aug. 2020, pp. 570–576. DOI: 10.1109/DSD51259.2020.00094. URL: https://ieeexplore.ieee.org/document/9217638 (visited on 01/07/2025) (cit. on p. 4).

[22] Prasanna Ravi, Vijaya Kumar Sundar, Anupam Chattopadhyay, Shivam Bhasin, and Arvind Easwaran. «Authentication Protocol for Secure Automotive Systems: Benchmarking Post-Quantum Cryptography». In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180847 (cit. on p. 4).

[23] Joppe W. Bos, Brian Carlson, Joost Renes, Marius Rotaru, Amber Sprenkels, and Geoffrey P. Waters. *Post-Quantum Secure Boot on Vehicle Network Processors*. Cryptology ePrint Archive, Paper 2022/635. 2022. URL: https://eprint.iacr.org/2022/635 (cit. on pp. 5, 91, 95, 96).

[24] Adam Dunkels and lwIP contributors. *lwIP: A Lightweight TCP/IP stack*. [Online; accessed 8. June 2026]. 2024. URL: https://savannah.nongnu.org/projects/lwip/ (cit. on pp. 5, 46).

[25] National Institute of Standards and Technology. *Recommendation for Key Management: Part 1 – General*. Tech. rep. NIST Special Publication 800-57 Part 1 Revision 5. National Institute of Standards and Technology, May 2020. DOI: 10.6028/NIST.SP.800-57pt1r5. URL: https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final (cit. on pp. 7, 8).

[26] Lov K. Grover. «A fast quantum mechanical algorithm for database search». In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 1996, pp. 212–219. DOI: 10.1145/237814.237866. URL: https://doi.org/10.1145/237814.237866 (cit. on p. 8).

[27] Gilles Brassard, Peter Høyer, and Alain Tapp. «Quantum algorithm for the collision problem». In: *Latin American Symposium on Theoretical Informatics*. Springer. 1998, pp. 163–169 (cit. on pp. 8, 9).

[28] Christof Zalka. «Grover's quantum searching algorithm is optimal». In: *Physical Review A* 60.4 (1999), pp. 2746–2751. DOI: 10.1103/PhysRevA.60.2746 (cit. on p. 9).

[29] National Institute of Standards and Technology (NIST). *Post-Quantum Cryptography Standardization: Security Evaluation Criteria*. [Online; accessed 24. May 2026]. 2016. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria) (cit. on pp. 9–11).

[30] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Tech. rep. FIPS 203. National Institute of Standards and Technology, Aug. 2024. DOI: 10.6028/NIST.FIPS.203. URL: https://doi.org/10.6028/NIST.FIPS.203 (cit. on p. 9).

[31] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard*. Federal Information Processing Standards Publication (FIPS) 204. Gaithersburg, MD: U.S. Department of Commerce, Aug. 2024. DOI: 10.6028/NIST.FIPS.204. URL: https://doi.org/10.6028/NIST.FIPS.204 (cit. on pp. 10–13, 20, 63, 80, 97, 100).

[32] Pierre-Alain Fouque et al. *Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU*. Submission to the NIST Post-Quantum Cryptography Standardization Process. Oct. 2021. URL: https://falcon-sign.info/falcon.pdf (cit. on pp. 10, 11, 21, 22, 80).

[33] Ray Perlner. *FIPS 206 Status Update*. Presentation, Computer Security Division, NIST. Slide deck previewing the planned FIPS 206 (FN-DSA) Initial Public Draft. 2025. URL: https://csrc.nist.gov/csrc/media/presentations/2025/fips-206-fn-dsa-(falcon)/images-media/fips_206-perlner_2.1.pdf (cit. on pp. 10, 63).

[34] National Institute of Standards and Technology (NIST). *Post-Quantum Cryptography: Additional Digital Signature Schemes*. [Online; accessed 2. Jun. 2026]. 2023. URL: https://csrc.nist.gov/projects/pqc-dig-sig (cit. on pp. 10, 11).

[35] Joppe W. Bos et al. *Hawk: A Fast, Compact and Floating-Point Free Post-Quantum Signature Scheme*. Submission to the NIST Additional Digital Signature Schemes Process (Round 2), version 1.1. Feb. 2025. URL: https://hawk-sign.info/hawk-spec-web.pdf (cit. on pp. 10, 11, 29, 33, 35, 63, 80, 97).

[36] National Institute of Standards and Technology (NIST). *Status Report on the Second Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. NIST IR 8610. National Institute of Standards and Technology, May 2026. DOI: 10.6028/NIST.IR.8610. URL: https://doi.org/10.6028/NIST.IR.8610 (cit. on pp. 10, 29, 30).

[37] J. Jonsson and B. Kaliski. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Tech. rep. RFC 8017. Defines the PSS encoding scheme and its security properties. RFC Editor, Nov. 2016. DOI: 10.17487/RFC8017. URL: https://www.rfc-editor.org/info/rfc8017 (cit. on p. 11).

[38] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Round 3)*. Submission to the NIST Post-Quantum Cryptography Standardization Process. Oct. 2021. URL: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf (cit. on p. 12).

[39] Amos Fiat and Adi Shamir. «How to Prove Yourself: Practical Solutions to Identification and Signature Problems». In: *Advances in Cryptology — CRYPTO '86*. Springer, 1986, pp. 186–194 (cit. on p. 15).

[40] Ronald Cramer. «Modular Design of Secure Signature Schemes». PhD thesis. CWI and University of Amsterdam, 1996 (cit. on p. 15).

[41] *Achieving a Root of Trust with Secure Boot in Automotive RH850 and R-Car Devices – Part 1 | Renesas*. en. URL: https://www.renesas.com/en/blogs/introduction-about-secure-boot-automotive-mcu-rh850-and-soc-r-car-achieve-root-trust-1 (visited on 01/07/2025) (cit. on pp. 38, 40–42).

[42] Christos Profentzas, Magnus Almgren, and Olaf Landsiedel. «Performance of Secure Boot in Embedded Systems». In: *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE, 2019 (cit. on p. 39).

[43] Andrew Regenscheid. *Platform Firmware Resiliency Guidelines*. Tech. rep. NIST Special Publication 800-193. National Institute of Standards and Technology, 2018. DOI: 10.6028/NIST.SP.800-193. URL: https://csrc.nist.gov/publications/detail/sp/800-193/final (cit. on pp. 41, 46, 47, 101, 102).

[44] Satoshi Yamanaka. *Achieving a Root of Trust with Secure Boot in Automotive RH850 and R-Car Devices — Part 2*. Renesas Electronics, engineering blog. Dec. 2021. URL: https://www.renesas.com/en/blogs/introduction-about-secure-boot-automotive-mcu-rh850-and-soc-r-car-achieve-root-trust-2 (cit. on p. 42).

[45] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. *Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4*. Cryptology ePrint Archive, Paper 2021/995. 2021. URL: https://eprint.iacr.org/2021/995 (cit. on p. 44).

[46] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. «Compact Dilithium Implementations on Cortex-M3 and Cortex-M4». In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (Dec. 2021), pp. 1–24. DOI: 10.46586/tches.v2021.i1.1-24. URL: https://doi.org/10.46586/tches.v2021.i1.1-24 (cit. on p. 44).

[47] Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. «Patterns for Secure Boot and Secure Storage in Computer Systems». In: *2010 Fifth International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2010, pp. 569–573. DOI: 10.1109/ARES.2010.110. URL: https://doi.org/10.1109/ARES.2010.110 (cit. on pp. 46, 47).

[48] *ISO 26262:2018: Road vehicles — Functional safety*. International Organization for Standardization. 2018. URL: https://www.iso.org/standard/68383.html (cit. on p. 47).

[49] AUTOSAR. *Specification of Secure Hardware Extension*. Tech. rep. AUTOSAR, 2022. URL: https://www.autosar.org/ (cit. on p. 48).

[50] Michele La Manna, Pericle Perazzo, Luigi Treccozzi, and Gianluca Dini. *Assessing the Cost of Quantum Security for Automotive Over-The-Air Updates*. 2021 (cit. on pp. 50, 96, 98, 100, 102).

[51] Claudius Pott, Philipp Jungklass, David Jacek Csejka, Thomas Eisenbarth, and Marco Siebert. «Firmware Security Module: A Framework for Trusted Computing in Automotive Multiprocessors». In: *Journal of Hardware and Systems Security* 5 (2021), pp. 103–113. DOI: 10.1007/s41635-021-00114-4 (cit. on pp. 51, 59, 98–101).

[52] The PQClean Project. *PQClean: Clean implementations of Post-Quantum Cryptography*. [Online; accessed 24. May 2026]. 2024. URL: https://github.com/pqclean/PQClean (cit. on pp. 63, 64).

[53] Daniel J. Bernstein and Tanja Lange. *SUPERCOP: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives*. [Online; accessed 8. June 2026]. 2024. URL: https://bench.cr.yp.to/supercop.html (cit. on pp. 63, 72).

[54] Thomas Pornin. *BearSSL*. [Online; accessed 24. May 2026]. 2018. URL: https://bearssl.org/ (cit. on pp. 63, 72).

[55] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Post-quantum cryptography on ARM Cortex-M4*. [Online; accessed 24. May 2026]. 2020. URL: https://github.com/mupq/pqm4 (cit. on pp. 65, 73, 95).

[56] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. «Evaluation of Lattice-Based Signature Schemes in Embedded Systems». In: *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, Dec. 2018, pp. 385–388. DOI: 10.1109/ICECS.2018.8617969 (cit. on p. 65).

[57] Michele Mosca and Marco Piani. *Quantum Threat Timeline Report 2025*. Tech. rep. Global Risk Institute, Mar. 2026. URL: https://globalriskinstitute.org/publication/quantum-threat-timeline-report-2025b/ (cit. on p. 94).